

UNIVERSITÉ NICE SOPHIA ANTIPOLIS – UFR Sciences

École Doctorale Sciences Fondamentales et Appliquées

THÈSE

pour obtenir le titre de
Docteur en Sciences
Spécialité MATHÉMATIQUES

présentée et soutenue par
Benedikt AHRENS

Initiality for Typed Syntax and Semantics

Thèse dirigée par **André HIRSCHOWITZ**
soutenue le 23 mai 2012

Membres du jury :

| | | | |
|----|--------------|-------------|-------------------------|
| M. | Pierre-Louis | CURIEN | Rapporteur et Examineur |
| M. | André | HIRSCHOWITZ | Directeur de thèse |
| M. | Marco | MAGGESI | Examineur |
| M. | Laurent | REGNIER | Rapporteur et Examineur |
| M. | Carlos | SIMPSON | Examineur |

Laboratoire Jean-Alexandre Dieudonné, Université de Nice, Parc Valrose, 06108 NICE

Abstract

In this thesis we give an algebraic characterization of the syntax and semantics of simply-typed languages. More precisely, we characterize *simply-typed binding syntax equipped with reduction rules* via a *universal property*, namely as the initial object of some category.

We specify a language by a *2-signature* (Σ, A) , that is, a signature on two levels: the *syntactic* level Σ specifies the sorts and terms of the language, and associates a sort to each term. The *semantic* level A specifies, through *inequations*, reduction rules on the terms of the language. To any given 2-signature (Σ, A) we associate a category of “models” of (Σ, A) . We prove that this category has an initial object, which integrates the terms freely generated by Σ and the reduction relation — on those terms — generated by A . We call this object the *programming language generated by* (Σ, A) .

Initiality provides an *iteration principle* which allows to specify translations on the syntax, possibly to a language over different sorts. Furthermore, translations specified via the iteration principle are by construction *type-safe* and *faithful with respect to reduction*.

To illustrate our results, we consider two examples extensively: firstly, we specify a double negation translation from classical to intuitionistic propositional logic via the category-theoretic iteration principle. Secondly, we specify a translation from PCF to the untyped lambda calculus which is faithful with respect to reduction in the source and target languages.

In a second part, we formalize some of our initiality theorems in the proof assistant Coq. The implementation yields a machinery which, when given a 2-signature, returns an implementation of its associated abstract syntax together with certified substitution operation, iteration operator and a reduction relation generated by the specified reduction rules.

Résumé

Dans cette thèse, on donne une caractérisation algébrique de la syntaxe et de la sémantique des langages simplement typés. Plus précisément, on caractérise la syntaxe simplement typée avec liaison de variables, équipée des règles de réduction, via une propriété universelle, à savoir comme l'objet initial d'une catégorie.

Nous spécifions un langage par une 2-signature (Σ, A) , c'est-à-dire, une signature à deux niveaux: le niveau *syntactique* Σ spécifie les types et les termes du langage, et associe un type à chaque terme. Le niveau *sémantique* A spécifie, via des *inéquations*, des règles de réduction sur les termes du langage. A chaque 2-signature (Σ, A) donnée on associe une catégorie des «modèles» de (Σ, A) . Nous démontrons que cette catégorie admet un objet initial, qui intègre les termes librement engendrés par Σ et la relation de réduction — sur ces termes — engendrée par A . Nous appelons cet objet le *langage engendré par* (Σ, A) .

Initialité fournit un *principe d'itération* qui permet de spécifier des traductions sur la syntaxe, possiblement vers un langage sur des types différents. De plus, les traductions qui sont spécifiées via ce principe d'itération sont *fidèles relativement au typage et la réduction*.

Afin d'illustrer nos résultats, nous considérons deux exemples en détail: premièrement, nous spécifions une traduction de la logique classique à la logique intuitioniste propositionnelle via le principe d'itération catégorique. Deuxièmement, nous spécifions une traduction de PCF au lambda calcul non-typé qui est fidèle par rapport aux réductions aux langages source et but.

Dans une deuxième partie, nous formalisons quelques uns de nos théorèmes d'initialité dans l'assistant de preuves Coq. L'implémentation apporte un mécanisme qui, étant donnée une 2-signature, rend une implémentation de sa syntaxe associée, équipée d'une opération de substitution certifiée, d'un opérateur d'itération et d'une relation de réduction engendrée par les règles de réduction spécifiées.

HELLO AND THANK YOU, ...

- André, for all your time and energy spent in working with me, and for advice on any subject
- Laurent and Pierre–Louis, for carefully reading this thesis and suggesting many improvements
- Carlos, for help and advice in various situations throughout my doctorate
- GGhh and Marco, for fun talk about science and stuff, and for receiving me in Florence
- Ingrid, for smoothing my path to Nice and further
- Jean–Marc and Julien, for tech support and guitar and linux talk
- LJAD and EDSFA administration crew, for making coping with administrative stuff a pleasure
- my Erasmus friends: Charline, Chiara, Daniela, Karo, Kerstin, Noémie, Ophélie, Sarah, Susanna, Tomke, GGhh, Henry, Marco, Martin, Nils, for going through the Erasmus experience with me
- my Florentine flatmates: Silvia & Carlo and Marzia & Antonio, and my office mates in stanza T1: Giulia, Loredana, John and Simone, for making me feel at home during my stay in Florence
- Amel, Audrey, Cindy, Ioana & Pierre, Irene & Marco, Julie & Sébastien, Laura & Benjamin, Monica, Nahla, Nancy, Olivia & Joan, Salima & Paul Eric, Sara, Silvia, Stéphanie, Vanessa, Ahed, Amine, Benjamin, Brahim, Brice, Giovanni, Hamad, Hugo, Luca, Marc, Nicolas, Osman, Raphaël, Rémy, Sarrage, Tolgahan, Tom V and Xavier, for giving me a nice time in Nice
- Julianna, for patiently answering my questions, and for writing a paper with me
- Debian and upstream, for providing the best operating system and tools, and, in particular, the Fossil SCM community
- Tobias and Michael, for news from Bayreuth and technical assistance
- Krissi, Nicki and “Volker”, for the fun time spent together
- Anne–Laure, for bearing with me, and family, for receiving me with such warmth
- Rike & Uwe, Vroni & Matze + Max, Feli & Clemi

RÉSUMÉ LONG

Dans cette thèse, on donne une caractérisation algébrique de la syntaxe et de la sémantique des langages simplement typés. Plus précisément, on caractérise la syntaxe simplement typée avec liaison de variables, équipée des règles de réduction, via une propriété universelle, à savoir comme l'objet initial d'une catégorie.

Sémantique Initiale

La *Sémantique Initiale* caractérise les termes d'un langage associés à une *signature* S comme l'objet initial d'une catégorie — dont on appellera les objets les *Sémantiques de S* —, ce qui fournit une définition concise de haut niveau de la syntaxe abstraite associée à S . Plus précisément, les ingrédients suivants sont utilisés:

Signature Une *signature* spécifie, de façon abstraite et concise, la syntaxe et la sémantique d'un langage.

Catégorie de Représentations A chaque signature S , on associe une catégorie de «modèles» de cette signature, que l'on appellera des *représentations de S* .

Initialité Dans cette catégorie de représentations de S , on exhibe l'objet initial, le *langage généré par S* .

Les motivations pour la Sémantique Initiale sont doubles: premièrement, la Sémantique Initiale fournit une définition catégorique — via une propriété universelle — de la syntaxe et de la sémantique engendrées librement par une signature. Deuxièmement, l'initialité donne lieu à un *opérateur d'itération* qui permet de spécifier de façon économique et conviviale des morphismes — *traductions* — de l'objet initial vers des autres langages.

Selon la «richesse» du langage qu'on veut spécifier, on a besoin d'une notion de signature adaptée et, en conséquence, d'une représentation de cette signature. Les caractéristiques que l'on considère dans cette thèse sont:

Liaison de Variables On considère des constructions liantes au *niveau des termes*, tels que l'abstraction lambda.

Typage On considère des systèmes de types *simples*, tels que le lambda calcul simplement typé et, via l'isomorphisme de Curry–Howard, la logique propositionnelle (cf. [Sect. 3.4](#)).

Réduction On considère de la sémantique sous forme de règles de réduction sur des termes, telles que la réduction bêta,

$$\lambda x.M(N) \rightsquigarrow M[x := N] .$$

Pour l'intégration de chacune des caractéristiques ci-dessus, les notions de signature et de représentation nécessitent d'être adaptées pour tenir compte de la quantité croissante d'information qui doit être fournie pour spécifier un langage.

Un de nos buts, c'est d'utiliser la Sémantique Initiale pour traiter la question suivante: nous voudrions traduire d'un langage à un autre — possiblement sur des ensembles de types différents —, en utilisant une construction universelle catégorique. Cette construction devrait prendre en compte le plus de «structure» possible. Par cela nous entendons que la traduction considérée devrait, par construction, être compatible, par exemple, avec le typage et réduction aux langages source et but.

Contributions

Dans cette thèse, nous donnons, via une propriété universelle, une caractérisation algébrique de la syntaxe simplement typée équipée d'une sémantique sous forme de règles de réduction. Plus précisément, étant donnée une *signature* — qui spécifie les types et les termes d'un langage — et des *inéquations* sur cette signature — qui spécifient des règles de réduction —, nous caractérisons les termes du langage associé à cette signature, équipés des règles de réduction selon les inéquations données, comme l'objet initial d'une catégorie des «modèles».

Notre point de départ est un travail sur l'initialité de la syntaxe non-typée effectué par Hirschowitz et Maggesi [HM07a], et sur son extension sur la syntaxe simplement typée par Zsidó [Zsi10]. Dans un premier temps nous étendons le théorème de Zsidó [Zsi10, Chap. 6] pour tenir compte des variations des types (cf. [Chapt. 3](#)). Puis, nous intégrons des règles de réduction dans le résultat d'initialité purement syntaxique d'Hirschowitz et Maggesi [HM07a], cf. [Chapt. 4](#). Finalement nous obtenons notre théorème principal, qui tient compte des variations des types ainsi que des règles de réduction, en combinant les deux résultats susmentionnés, cf. [Chapt. 5](#).

De plus, pour le cas non-typé, nous fournissons une preuve formalisée dans l'assistant de preuves Coq de notre résultat, ce qui donne un mécanisme qui, étant donnée une signature pour des termes et un ensemble d'inéquations, produit la syntaxe abstraite associée à cette signature, équipée de la relation de réduction engendrée par les inéquations. Pour le cas simplement typé, nous formalisons l'instance de notre résultat principal (cf. [Thm. 5.21](#)) pour la signature du langage de programmation PCF [Plo77].

Nous décrivons maintenant nos contributions en détail:

Une variante du théorème de Zsidó

Dans sa thèse, [Zsi10, Chap. 6], Zsidó démontre un théorème d'initialité pour la syntaxe abstraite associée à une signature simplement typée. Pourtant, les modèles qu'elle considère, dont la syntaxe abstraite est initiale, sont tous des modèles sur le même ensemble de types. Ainsi, le principe d'itération obtenu par initialité ne permet pas la spécification d'une traduction vers un langage sur un ensemble différent de types. Nous adaptons son théorème en introduisant des *signatures typées*. Une signature typée (S, Σ) spécifie un ensemble de *types* via une signature algébrique S , ainsi qu'un ensemble de *termes* simplement typés sur ces types via une signature de termes Σ sur S .

Une représentation R d'une telle signature typée est alors donnée par une représentation de sa signature S pour les types dans un ensemble $T = T_R$ ainsi qu'une représentation de Σ dans une monade — aussi appelée R — sur la catégorie Set^T . Un morphisme de représentations $P \rightarrow R$ est constitué d'un morphisme f entre les représentations de S sous-jacentes, et d'un morphisme de représentations de Σ qui est compatible dans un sens approprié avec la «traduction des types» f . Nous démontrons que la catégorie des représentations de (S, Σ) ainsi définie admet un objet initial, qui intègre les types librement engendrés par S et les termes librement engendrés par Σ , typés sur les types de S . Notre définition de morphismes assure que, pour toute traduction spécifiée par le principe d'itération, la traduction des termes est compatible avec la traduction des types par rapport au typage des langages source et but.

Syntaxe non-typée et règles de réduction

Pour intégrer des règles de réduction à nos résultats d'initialité, nous définissons la notion de *2-signature*. Une 2-signature (Σ, A) est donnée par une (1-)signature Σ qui spécifie les termes d'un langage, et un ensemble A d'*inéquations* sur Σ . Intuitivement, chaque inéquation spécifie une règle de réduction, par exemple la règle bêta.

Les *modèles* — ou *représentations* — d'une telle 2-signature sont construits à partir des *monades relative* et des *modules sur des monades relatives*: étant donnée une 1-signature Σ , nous définissons une représentation de Σ comme étant donnée par une monade relative sur le foncteur approprié $\Delta : \text{Set} \rightarrow \text{Pre}$ (cf. Def. 2.13), accompagnée d'un morphisme de modules (sur des monades relatives) approprié pour chacune des arités de Σ . Étant donné un ensemble A d'inéquations sur Σ , nous définissons un prédicat de satisfaction pour les modèles de Σ ; nous appelons *représentation de (Σ, A)* chaque représentation de Σ qui satisfait chacune des inéquations de A . Ce prédicat spécifie une sous-catégorie pleine de la catégorie des représentations de Σ . Nous appelons cette sous-catégorie la *catégorie des représentations de (Σ, A)* . Nous démontrons que cette catégorie admet un objet initial, qui est construit en équipant la représentation initiale de Σ — donnée par les termes librement engendrés par Σ — d'une relation de réduction appropriée engendrée par les inéquations de A .

Avec ce théorème d'initialité de (Σ, A) nous obtenons un nouveau principe d'itération, et chaque traduction qui est spécifiée via ce principe est, par construction, compatible avec la relation de réduction aux langages source et but.

Théorème principal: Systèmes de types simples et réductions

Finalement, nous combinons les deux théorèmes susmentionnés pour obtenir un résultat d'initialité qui tient compte de notre exemple principal, une traduction de PCF vers le lambda calcul non-typé. Plus précisément, nous définissons une *2-signature* comme étant donnée par une signature typée (S, Σ) , accompagnée d'un ensemble A d'inéquations sur (S, Σ) qui spécifie des règles de réduction.

Nous définissons une catégorie de représentations (S, Σ) et nous démontrons que cette catégorie admet un objet initial. Cette représentation initiale intègre les types et les termes librement engendrés par (S, Σ) , les termes étant équipés d'une relation de réduction engendrée par les inéquations de A .

Une implémentation sur machine pour la spécification de syntaxe et sémantique

Les théorèmes susmentionnés sont faits pour être implémentés dans un assistant de preuves. Une telle implémentation permet la spécification de syntaxe et règles de réduction via des 2-signatures, fournissant un mécanisme fortement automatisé pour produire de la syntaxe équipée d'une substitution certifiée et d'un principe d'itération.

Nous démontrons le théorème pour syntaxe non-typée avec règles de réduction décrit en haut dans l'assistant de preuves Coq [Coq10]. Comme illustration, nous décrivons comment obtenir le lambda calcul avec réduction bêta via initialité.

De plus, nous formalisons une instance du théorème principal, également en Coq. Plus précisément, nous définissons la catégorie des représentations de la signature typée de PCF avec des réductions et nous démontrons que cette catégorie admet un objet initial. Après, nous donnons une représentation de cette signature dans la monade relative du lambda calcul avec réduction bêta ULC_β , ce qui fournit une traduction de PCF vers ULC. Des instructions sur comment obtenir le code source complet de notre bibliothèque Coq sont disponible sur

<http://math.unice.fr/laboratoire/logiciels>.

INTRODUCTION

Motivation: Traductions de PCF vers ULC

Comme exemple introductif, on considère des traductions de PCF, introduit par Plotkin [Plo77], vers le lambda calcul de Church [Chu36]. Une description détaillée des deux langages est donnée dans Appx. A. Ces deux langages sont paradigmatiques au sens où PCF peut être vu comme un langage de haut niveau, équipé d'un système de types, tandis que le lambda calcul représente un langage non-typé de bas niveau.

Nous spécifions une application f de l'ensemble de termes de PCF vers le lambda calcul comme dans Fig. 1.1 (cf. [Pho93]), avec une fonction g des constantes de PCF vers des lambda termes, e.g., $g(\mathbf{T}) := \lambda xy.x$, et constantes du lambda calcul, e.g.,

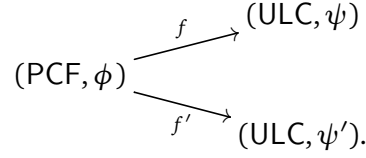
$$\Theta := (\lambda x.\lambda y.(y(xxy))) (\lambda x.\lambda y.(y(xxy))) \quad (\text{Turing fixed point combinator}) \text{ and} \\ \Omega := (\lambda x.xx)(\lambda x.xx) .$$

Bien entendu, différentes traductions existent; par exemple, on pourrait traduire **Fix** vers un combinateur de point fixe différent.

Dans cette thèse on présente un cadre catégorique pour la spécification des tels traductions d'un langage vers un autre. Les challenges sont:

- les ensembles de types différents des langages source et but et
- intégrer la compatibilité de telles traductions avec la structure — substitution et réduction — des langages source et but.

Nous définissons une catégorie dans laquelle les langages comme PCF et ULC sont des objets, et dans laquelle une traduction comme décrite plus haut est un morphisme $f : \text{PCF} \rightarrow \text{ULC}$. Plus précisément, dans la catégorie qu'on construit, la traduction f est un *morphisme initial* $f : \text{PCF} \rightarrow \text{ULC}$, c'est-à-dire, sa source PCF est l'objet initial. Il y a plusieurs traductions possibles de PCF vers ULC, et le morphisme $f : \text{PCF} \rightarrow \text{ULC}$ ne peut pas être initial dans une catégorie où les objets ne sont «que» des langages — autrement on aurait $f = f'$ pour toute traduction $f' : \text{PCF} \rightarrow \text{ULC}$. Donc les objets dans la catégorie qu'on construit sont des langages avec de la structure de plus, qui permet de distinguer des morphismes initiaux $f, f' : \text{PCF} \rightarrow \text{ULC}$,



Dans cette catégorie, initialité de (PCF, ϕ) donne le principe d'itération suivant: spécifier une traduction itérative $f : PCF \rightarrow ULC$ est équivalent à spécifier la “structure additionnelle” ψ du lambda calcul ULC.

Une question naturelle est si — ou mieux, dans quel sens — la traduction f spécifiée dans Fig. 1.1 est compatible avec les réductions respectives des langages source et but. Phoa [Pho93] répond à cette question; en particulier, la traduction f est *fidèle* au sens que

$$t \rightarrow_{PCF} t' \text{ implique } f(t) \rightarrow_{\beta} f(t') .$$

Dans cette thèse nous fournissons un cadre catégorique qui permet de spécifier, via une propriété universelle, de telles traductions fidèles entre des langages avec liaison sur des ensembles de types différents.

Exemple: Axiomes de Peano

On introduit la notion de *signature* et *représentation* à l'exemple des nombres naturels; on donne la signature des nombres naturels ainsi que la catégorie des représentations associée. Comme *signature*, nous considérons l'application suivante:

$$\mathcal{N} := \{z \mapsto 0, \quad s \mapsto 1\} .$$

Les nombres naturels sont construit à partir de deux constructeurs, notamment un opérateur d'arité 0, disons, z , — la constante zéro — ainsi qu'un opérateur unaire, disons, s — la fonction successeur.

Une *représentation* de la signature \mathcal{N} est donnée par un triplet (X, Z, S) d'un ensemble X avec une constante $Z \in X$ et une opération unaire $S : X \rightarrow X$. Un morphisme vers un autre triplet (X_0, Z_0, S_0) est donné par une application $f : X \rightarrow X_0$ telle que

$$f(Z) = Z_0 \quad \text{et} \quad f \circ S = S_0 \circ f .$$

Cette catégorie admet un *objet initial* $(\mathbb{N}, \text{Zero}, \text{Succ})$ donné par les nombres naturels \mathbb{N} équipés de la constante $\text{Zero} = 0$ et de l'application successeur $\text{Succ} : \mathbb{N} \rightarrow \mathbb{N}$.

Liaison des Variables

Les techniques suivantes sont utilisées fréquemment pour modéliser la liaison des variables:

- Syntaxe nominelle utilisant l'abstraction nommée (\mathbb{A} étant un ensemble d'*atomes*), e.g.,

$$\lambda : [\mathbb{A}]T \rightarrow T$$

- Higher-Order Abstract Syntax (HOAS), e.g.,

$$\lambda : (T \rightarrow T) \rightarrow T$$

et sa variante *faible*, e.g.,

$$\lambda : (\mathbb{A} \rightarrow T) \rightarrow T$$

- Nested Data Types comme présentés par [BM98], e.g.,

$$\lambda : T(X + 1) \rightarrow T(X)$$

L'encodage via nested data types est différent des autres techniques au sens qu'ici, l'ensemble des termes T est paramétrisé par un *contexte*. Donc $T(X)$ dénote l'ensemble des termes du langage T avec des variables libres dans l'ensemble X . L'ensemble $X + 1$ correspond à un *contexte élargi* d'une variable libre additionnelle, qui sera liée par le constructeur lambda.

Exemple: Nous représentons le lambda calcul comme un *nested data type*: considérons le type inductif $\text{ULC} : \text{Set} \rightarrow \text{Set}$:

Inductive $\text{ULC} (V : \text{Type}) : \text{Type} :=$
 | Var : $V \rightarrow \text{ULC } V$
 | Abs : $\text{ULC} (\text{option } V) \rightarrow \text{ULC } V$
 | App : $\text{ULC } V \rightarrow \text{ULC } V \rightarrow \text{ULC } V$.

Pour la syntaxe avec liaison, les arités doivent donner de l'information sur les liaisons du constructeur associé. Nous spécifions les arités avec des listes de nombres naturels. La longueur d'une liste spécifie le nombre d'arguments d'un constructeur, et sa composante i donne le nombre de variables que le constructeur lie dans l'argument i . La signature Λ de ULC est donnée par

$$\Lambda := \{\text{app} : [0, 0] , \quad \text{abs} : [1]\} .$$

L'application $V \mapsto \text{ULC}(V)$ est functorielle: pour $f : V \rightarrow W$, l'application $\text{ULC}(f) : \text{ULC}(V) \rightarrow \text{ULC}(W)$ *renomme* chaque variable libre $v \in V$ d'un terme par $f(v)$, ce qui donne un terme avec des variables libres dans W . Alors, la signature Λ doit être représentée dans des foncteurs $F : \text{Set} \rightarrow \text{Set}$ au lieu des ensembles, et on considère des transformations naturelles au lieu des applications.

Substitution

Nous souhaitons intégrer le plus de structures possible dans notre catégorie de «modèles». Une de ces structures est la *substitution sans capture* des variables libres. Pour cela, nous ne considérons pas des foncteurs simples $F : \text{Set} \rightarrow \text{Set}$, mais des *monades* sur la catégorie Set des ensembles. Une monade est un foncteur équipé de structure additionnelle, que l'on explique en utilisant l'exemple du lambda calcul. L'application $V \mapsto \text{ULC}(V)$ vient avec une opération de substitution simultanée sans capture: soient V and W deux ensembles (de variables) et f une application $f : V \rightarrow \text{ULC}(W)$. Etant donné un lambda terme $t \in \text{ULC}(V)$, on remplace chaque variable libre $v \in V$ dans t par son image sous f , ce qui donne un terme $t' \in \text{ULC}(W)$. De plus, nous considérons le constructeur Var_V comme une application “variable–comme–terme”, indexée par un ensemble de variables V ,

$$\text{Var}_V : V \rightarrow \text{ULC}(V) .$$

Altenkirch et Reus [AR99] observent que la structure de monade capture ces deux opérations et leurs propriétés: substitution et variable–comme–termes font de ULC une monade sur la catégorie des ensembles.

La structure de monade de ULC devrait être compatible dans un sens avec les constructeurs Abs et App de ULC : *substitution distribuée sur les constructeurs*. Pour capturer cette distributivité, Hirschowitz et Maggesi [HM07a] considèrent des *modules sur une monade* (cf. Def. 2.43) — qui généralisent la substitution monadique —, et des morphismes de modules — qui sont des transformations naturelles qui sont compatibles avec la substitution de modules. En effet, les applications

$$\begin{aligned} \text{ULC} : V &\mapsto \text{ULC}(V) , \\ \text{ULC}' : V &\mapsto \text{ULC}(V + 1) \text{ et} \\ \text{ULC} \times \text{ULC} : V &\mapsto \text{ULC}(V) \times \text{ULC}(V) \end{aligned}$$

sont des application sous-jacentes de tels modules (cf. Exs. 2.45, 2.46), et les constructeurs Abs et App sont des morphismes de modules (cf. Exs. 2.47, 2.74).

Types

Des *systèmes de types* existent avec des caractéristiques variées, de la syntaxe simplement typée à la syntaxe avec des types dépendants, polymorphisme etc. Par syntaxe simplement typée nous entendons une syntaxe non-polymorphe dont l'ensemble de types est indépendant de l'ensemble des termes, c'est-à-dire les constructeurs de types ne prennent que des types comme arguments.

Dans des systèmes de types plus sophistiqués, les types peuvent dépendre des termes, ce qui amène à des définitions plus complexes d'arité et de signature. Ce travail-ci ne

traite que les langages simplement typés, comme le lambda calcul simplement typé ou PCF. Nous appellerons l'ensemble de types sous-jacent les *types objet*.

Le but du typage est de *classifier* les termes selon des critères. Par exemple, on pourrait se demander si un terme est de type fonction, et ainsi peut être appliqué à un autre terme. Une fois qu'une telle classification est mis en place, on peut utiliser l'information de typage pour filtrer les termes selon leurs types, pour ne choisir que les termes avec le type désiré.

Une façon d'ajouter des types serait de les intégrer dans les termes comme dans « $\lambda x : \mathbb{N}.x + 4$ ». Par contre, pour les *systèmes de types simples* on peut séparer les univers des types et des termes et considérer le typage comme une application des termes vers les types, ainsi donnant une structure simple mathématique au typage.

Comment peut-on assurer que nos termes sont bien typés ? Bien qu'on sépare les types des termes, on voudrait maintenir une intégration forte du typage dans le processus de construction des termes, pour éviter de construire des termes mal typés. La séparation des termes et des types semble contredire ce but. La réponse est de ne pas considérer qu'un ensemble de termes avec une application de typage vers l'ensemble, disons, T de types, mais une *famille d'ensembles*, indexée par l'ensemble T de types objet. Les constructeurs de termes peuvent ainsi choisir quels termes ils accepteront comme argument. Nous considérons aussi les variables libres comme étant équipées d'un type objet. Autrement dit, nous ne considérons pas des termes sur un ensemble de variables, mais sur une famille d'ensembles de variables, indexée par l'ensemble des types objet. Encore autrement dit, nous considérons un contexte comme donné par une famille $(V_t)_{t \in T}$ d'ensembles, d'où $V_t := V(t)$ est l'ensemble de variables de type t . Nous illustrons notre point de vue à l'aide de l'exemple du lambda calcul simplement typé TLC:

Exemple: Soit

$$T_{\text{TLC}} ::= * \mid T_{\text{TLC}} \rightsquigarrow T_{\text{TLC}}$$

l'ensemble de types du lambda calcul simplement typé. L'ensemble des lambda termes avec des variables libres dans V est donné par la famille inductive suivante:

Inductive TLC $(V : T \rightarrow \text{Type}) : T \rightarrow \text{Type} :=$
 | Var : forall t, V t \rightarrow TLC V t
 | Abs : forall s t TLC (V + s) t \rightarrow TLC V (s \rightsquigarrow t)
 | App : forall s t, TLC V (s \rightsquigarrow t) \rightarrow TLC V s \rightarrow TLC V t.

d'où $V + s := V + \{s\}$ est l'extension du contexte par une variable de type $s \in T_{\text{TLC}}$ — la variable qui sera liée par le constructeur Abs(s, t). Les variables s et t prennent des valeurs dans l'ensemble T_{TLC} des types. La signature du lambda calcul simplement typé est donnée dans Exs. 3.23 et 3.47. Le paragraphe précédent sur les monades et modules s'applique au lambda calcul simplement typé quand on remplace les ensembles par des familles d'ensembles indexées par T_{TLC} : le lambda calcul simplement typé peut être

équipé d'une structure de monade (cf. Ex. 2.37)

$$\text{TLC} : \text{Set}^{T_{\text{TLC}}} \rightarrow \text{Set}^{T_{\text{TLC}}} .$$

Les constructeurs de TLC sont des morphismes de modules (cf. Exs. 2.61, 2.56, 2.60).

Cette méthode de définir précisément les termes bien typés en les organisant dans une famille d'ensembles paramétrisée par les types objet s'appelle *typage intrinsèque* [BHKM11] — l'opposé du *typage extrinsèque*, où d'abord on définit un ensemble de termes *bruts*, qui est filtré après via un prédicat de typage. Le typage intrinsèque délègue le typage objet au système de type du méta langage, comme Coq dans Ex. 1.3. Ainsi, le système de types méta (e.g. Coq) trie les termes mal typés automatiquement: écrire un tel terme donne une erreur de type au niveau méta.

De plus, l'encodage intrinsèque vient avec un principe de récursion plus conviviale; une application vers un autre système de types peut être donnée en spécifiant son image sur les termes bien typés. En utilisant le typage extrinsèque, une application sur les termes serait spécifiée sur l'ensemble des termes *bruts*, y compris les termes mal typés, ou seulement sur les termes bien typés en donnant un argument propositionnel de plus qui exprime le fait que le terme soit bien typé. Benton et al. donnent une explication détaillée du typage intrinsèque [BHKM11].

Réductions

La *sémantique* d'un langage de programmation décrit comment des logiciels de ce langage sont *évalués*. Pour les langages fonctionnels comme on les considère dans cette thèse, l'évaluation est faite par des *réductions*. Par exemple, l'évaluation du terme $7 + 5$ d'un langage arithmétique vers sa valeur 12 est faite en une série de réductions, dont la forme précise dépend de la sémantique du langage. Des *règles* typiques, qui spécifient comment des termes réduisent, sont données dans Sect. A.2 pour les langages du lambda calcul et PCF.

Etant donné un ensemble A de règles de réduction, on peut considérer la relation engendrée par ces règles. Plus précisément, suivant Barendregt et Barendsen [BB94], nous considérons plusieurs *clôtures* de ces règles:

Propagation dans des sous-termes Une relation R est appelé *compatible* si elle est close sous propagation dans des sous-termes, i.e. si pour tout constructeur f d'arité n et tout $i \leq n$,

$$M \rightsquigarrow_R N \Rightarrow f(x_1, \dots, x_{i_1}, M, x_{i+1}, \dots, x_n) \rightsquigarrow_R f(x_1, \dots, x_{i_1}, N, x_{i+1}, \dots, x_n) .$$

Réduction Une relation R est une *relation de réduction* si elle est compatible, réflexive et transitive.

Equivalence Une relation R est une *congruence* si elle est une relation d'équivalence compatible.

A l'ensemble A de règles nous associons trois relations engendrées par A , qui sont les relations les plus petites contenant A et étant une relation compatible, une relation de réduction et une relation d'équivalence, respectivement. Nous écrivons ces relations, dans cet ordre, par \rightarrow_A , \twoheadrightarrow_A and $=_A$, respectivement.

Dans cette thèse nous considérons la *relation de réduction* engendrée par un ensemble de règles. Par rapport à la congruence, il lui manque une règle de *symmetrie*, ce qui, bien qu'adéquat pour le raisonnement mathématique, donne lieu à une relation trop grossière du point de vue du *calcul*. Comme l'écrit Girard [GTL89], tandis que la congruence engendrée par A accentue le point de vue *statique* des mathématiques, la relation de réduction associée à A accentue le point de vue *dynamique* du calcul.

Afin de tenir compte des réductions, nous considérons des foncteurs et monades dont le codomaine n'est pas la catégorie des (familles d') ensembles, mais des (familles d') *ensembles préordonnés*. La définition de monade demande du foncteur sous-jacent d'être un endofoncteur, mais nous ne voudrions pas considérer des contextes préordonnés — quelle serait la signification de ce préordre ? La restriction à des endofoncteurs a été abolie par Altenkirch et al. [ACU10] en introduisant les monades *relatives*. Une monade relative est donnée par un foncteur — pas nécessairement endo — accompagné de deux opérations très similaires aux opérations monadiques variables-comme-termes et substitution. Nous considérons ainsi, par exemple, le lambda calcul comme une monade relative qui associe, à chaque *ensemble* X de variables, un *ensemble préordonné de lambda termes* $(\text{ULC}(X), \twoheadrightarrow_\beta)$, où le préordre sur $\text{ULC}(X)$ est donné par la relation de réduction \twoheadrightarrow_β engendrée par la règle bêta de Disp. (A.2.1), cf. Ex. 2.85.

CONCLUSIONS ET TRAVAUX ULTÉRIEURS

Nous résumons les contributions de cette thèse et abordons des travaux ultérieurs.

Contributions

Nous avons démontré un résultat d’initialité pour de la syntaxe simplement typée, équipée des règles de réduction. Le principe d’itération catégorique obtenu par la propriété universelle d’initialité est suffisamment général pour permettre la spécification de traductions de la représentation des termes vers des langages typés sur des ensembles *différents* des types.

Nous avons caractérisé la syntaxe liante avec des réductions — par exemple, le lambda calcul avec la réduction bêta — comme une monade relative sur le foncteur Δ (cf. [Ex. 2.85](#)), ce qui n’encode pas seulement des propriétés de commutativité de la substitution, mais également sa monotonie dans l’argument d’ordre premier. Une autre propriété de monotonie pour l’argument d’ordre supérieur peut être assurée par un renforcement approprié de la définition de monade relative dans un contexte 2-catégorique, cf. [Rem. 2.86](#). Nous avons également transféré la définition de *module* sur une monade et plusieurs constructions de modules vers des modules sur les monades relatives.

Ensuite, nous avons démontré plusieurs théorèmes dans l’assistant de preuves Coq: premièrement, nous avons implémenté le théorème d’initialité de Zsidó [[Zsi10](#), Chap. 6], résumé dans ce travail pour référence dans [Sect. 3.2](#). Deuxièmement, nous avons démontré le théorème de [Chapt. 4](#), fournissant un outil qui, étant donnée une 2-signature (S, A) , génère la syntaxe associée à S , équipée de la relation de réduction engendrée par les inéquations de A . Troisièmement, nous avons démontré une instance de notre théorème principal, [Thm. 5.21](#) de [Chapt. 5](#), pour la 2-signature particulière du langage de programmation PCF, équipé des règles de réduction comme dans [Fig. A.4](#). La représentation de la signature de PCF dans la monade du lambda calcul non typé avec réduction bêta donne une traduction exécutable de PCF vers ULC qui est certifiée d’être compatible avec la substitution et la réduction des langages source et but.

Travaux Ultérieurs

Désormais, nous espérons démontrer et implémenter des théorèmes d'initialité pour des systèmes de types plus riches. En particulier, on voudrait prendre en compte des types *dépendants* et le *polymorphisme*, deux étapes importantes vers des logiciels certifiés et réutilisation de code, respectivement.

De plus, la modélisation de la sémantique devrait être améliorée pour permettre le raisonnement sur des propriétés importantes telles que la terminaison.

Comme susmentionné, l'implémentation des résultats d'initialité dans un assistant de preuves peut servir comme un cadre pour la recherche sur des langages de programmation et des logiques. Pour cette raison, nous envisageons l'implémentation dans un assistant de preuves de [Thm. 5.21](#) en toute généralité.

On présente ces aspects en détail:

Modélisation de réduction plus nuancée Etant donnée une 2-signature (une signature avec un ensemble d'inéquations), les modèles pour cette 2-signature étaient jusqu'à maintenant principalement des foncteurs qui associent, à chaque ensemble «de variables» un ensemble préordonné — intuitivement un modèle des «termes» sur l'ensemble des variables¹. Le préordre \leq sur un tel modèle correspond à la relation de réduction sur ce modèle, c'est-à-dire le «terme» t réduit vers t' si et seulement si $t \leq t'$.

La modélisation des réductions via des *préordres* peut être considérée comme étant trop grossière à plusieurs égards:

- des réductions différentes peuvent amener d'un terme vers un autre. Par contre, l'utilisation des préordres pour la modélisation des réductions ne permet pas de distinguer deux réductions de même source et but.
- La règle de réflexivité codée en dur rend difficile le raisonnement sur la *normalisation* — en particulier, la *terminaison*.

Au lieu de considérer des ensembles *préordonnés* (indexés par des ensembles de variables libres) comme des modèles d'une 2-signature, il serait intéressant de considérer une structure qui permet un traitement plus nuancé de réduction, comme par exemples les graphes ou les catégories. Autrement dit, on pourrait construire des modèles d'une 2-signature à partir des monades relatives vers la catégorie des *graphes* ou (petites) *catégories*. En utilisant cette nouvelle définition de modèle, on pourrait envisager de démontrer un théorème d'initialité analogue à celui déjà démontré, et d'utiliser la structure de plus obtenue en travaillant avec des graphes ou des catégories pour raisonner sur les propriétés mentionnées plus haut.

¹On ignore le cas typé pour l'instant, qui est analogue.

Inéquations, Syntaxiquement Fiore et Hur [FH10] développent une théorie syntaxique d'équations sur une signature d'ordre supérieure, ce qui permet de prouver sûreté et complétude par rapport aux modèles de la signature et aux équations. Des techniques pareilles devraient permettre de présenter nos inéquations de façon syntaxique. En plus du but évident de sûreté et complétude, une telle présentation syntaxique faciliterait aussi la spécification des réductions dans l'implémentation en Coq: en particulier, il serait possible de spécifier des réductions sans aucune connaissance des concepts catégoriques.

Un but minimal, ce serait d'avoir un data type — qui dépend de la 1-signature sous-jacent — qui permet de spécifier les demi-équations habituelles, principalement obtenues par la substitution et en composant des arités, p.ex. $\text{app} \circ (\text{abs} \times \text{id})$. A un terme de ce data type on pourrait associer une famille de morphismes de modules, qui forment le carrier d'une demi-équation: les propriétés algébriques (d'être un morphisme de modules, ce qui correspond à la compatibilité entre substitution et meta-substitution dans [FH10]) pourraient être prouvées une fois pour tout par récurrence.

Systèmes de types plus sophistiqués Les nouveaux langages de programmation sont équipés de systèmes de types de plus en plus sophistiqués: des *types dépendants* permettent d'assurer des propriétés des résultats d'une fonction et ainsi la composition fiable des fonctions. Le *polymorphisme* permet la réutilisation de code dans des situations diverses. Une caractérisation algébrique de tels systèmes de types sophistiqués avec liaison de variables par une propriété universelle n'existe pas encore. Nous espérons généraliser nos résultats d'initialité pour prendre en compte ces systèmes de types.

Une classe plus large d'arités Les théorèmes d'initialité jusqu'à maintenant prennent en compte des arités, c'est-à-dire des constructeurs de termes, de nature plutôt simple: les seules opérations considérées sont le produit — pour des constructeurs qui prennent *plusieurs* arguments — et l'extension de contexte, pour modéliser la liaison de variables.

On devrait tenir compte des constructeurs de termes plus généraux. Hirschowitz et Maggesi [HM12] ont introduit une notion d'arité renforcée qui permet, par exemple, de traiter un constructeur d'aplatissement $\mu : T \circ T \rightarrow T$. Finalement, nous espérons trouver un critère *simple* très général pour des arités et des signatures pour lesquelles un modèle initial peut être construit.

Un outil de recherche certifié Les résultats obtenus devraient — comme on l'a déjà fait pour la syntaxe non typée avec réductions — être implémentés dans un assistant de preuves tel que Coq. Ainsi, un théorème d'initialité peut être utilisé comme un outil pratique pour faire facilement des expériences avec des langages

différents. Changer un langage correspondrait à simplement changer sa signature spécifiante, et toutes les données et propriétés telles que la substitution certifiée et le principe d'itération, mais également des réductions, seraient fournies par le système. Pour cette implémentation sur la machine et pour avoir des règles de réduction appropriées, nous souhaitons aussi obtenir, de façon automatique, une *fonction* de réduction r en plus de la relation de réduction. Cette fonction de réduction pourrait ainsi être validée par rapport à la relation au sens où l'on pourrait démontrer que pour chaque terme t , on a $t \leq r(t)$.

CONTENTS

| | |
|--|---------------|
| 1. INTRODUCTION | 1 |
| 1.1. Motivation: Translations from PCF to ULC | 1 |
| 1.2. Initial Semantics | 3 |
| 1.2.1. Example: Peano Axioms | 4 |
| 1.2.2. Initial Algebras | 5 |
| 1.2.3. Adding Variable Binding | 6 |
| 1.2.4. Adding Substitution | 7 |
| 1.2.5. Adding Types | 8 |
| 1.2.6. Adding Reductions | 9 |
| 1.3. Contributions | 11 |
| 1.3.1. Extended Initiality for Varying Sorts | 11 |
| 1.3.2. Integrating Reduction Rules | 12 |
| 1.3.3. Main Theorem: Initiality for Simply-Typed Syntax with Reduction | 13 |
| 1.3.4. A Computer Implementation for Specifying Syntax and Semantics | 13 |
| 1.4. Synopsis | 13 |
| 1.5. Related Work | 16 |
| 1.5.1. Translations from PCF | 16 |
| 1.5.2. Initial Semantics | 16 |
| 1.5.3. Formalization of Syntax | 19 |
| 1.5.4. Published Work | 19 |
| I. THEORY | 21 |
| 2. CATEGORY-THEORETIC CONSTRUCTIONS | 23 |
| 2.1. Categories, Functors & Transformations | 23 |
| 2.1.1. Two Definitions of Categories | 23 |
| 2.1.2. Functors & Natural Transformations | 25 |
| 2.1.3. More Examples, Notations | 28 |
| 2.2. Monads & Modules | 31 |
| 2.2.1. Definitions | 31 |
| 2.2.2. Constructions on Monads and Modules | 35 |
| 2.2.3. Monads on Set Families | 36 |
| 2.3. Alternative Definitions for Monads & Modules | 39 |

| | |
|--|------------|
| 2.4. Relative Monads and Modules | 44 |
| 2.4.1. Definitions | 44 |
| 2.4.2. Constructions on Relative Monads and Modules | 52 |
| 2.4.3. Derivation & Fibre | 53 |
| 3. SIMPLE TYPE SYSTEMS | 57 |
| 3.1. Signatures for Types | 57 |
| 3.2. Zsidó's Theorem Reviewed | 60 |
| 3.2.1. Signatures for Terms | 60 |
| 3.2.2. Representations | 65 |
| 3.2.3. Initiality | 66 |
| 3.3. Extending Zsidó's Theorem to Varying Types | 66 |
| 3.3.1. Signatures for Types & Terms | 67 |
| 3.3.2. Representations of Typed Signatures | 74 |
| 3.3.3. Initiality | 76 |
| 3.4. Logics and Logic Translations | 79 |
| 3.4.1. Signatures of Classical and Intuitionistic Logic | 79 |
| 3.4.2. Translation via Initiality | 80 |
| 3.4.3. Remarks | 81 |
| 4. REDUCTIONS FOR UNTYPED SYNTAX | 83 |
| 4.1. 1-Signatures | 84 |
| 4.2. Representations of 1-Signatures | 87 |
| 4.3. Inequations | 90 |
| 4.4. Initiality for 2-Signatures | 93 |
| 5. SIMPLE TYPE SYSTEMS WITH REDUCTIONS | 99 |
| 5.1. 1-Signatures | 99 |
| 5.2. Representations of 1-Signatures | 101 |
| 5.3. Inequations | 102 |
| 5.4. Initiality for 2-Signatures | 105 |
| II. COMPUTER IMPLEMENTATION | 111 |
| 6. FORMALIZING CATEGORY THEORY IN Coq | 113 |
| 6.1. About the Proof Assistant Coq | 113 |
| 6.2. Formalizing Algebraic Structures | 114 |
| 6.3. Formalizing Categories | 117 |
| 6.3.1. Which Definition to Formalize — Dependent Hom-Sets? | 117 |
| 6.3.2. Setoidal Equality on Morphisms | 118 |
| 6.3.3. Coq Setoids and Setoid Morphisms | 118 |

| | |
|---|------------|
| 6.3.4. Coq Implementation of Categories | 119 |
| 6.3.5. The Categories of Interest | 120 |
| 6.3.6. Initial Objects | 121 |
| 6.3.7. Interlude on the Program feature | 121 |
| 6.3.8. Retyping and Option | 122 |
| 6.4. Monads, Modules and their Morphisms | 123 |
| 6.5. Relative Monads, Formalized | 125 |
| 7. FORMALIZATION OF ZSIDÓ'S THEOREM | 127 |
| 7.1. Signatures & Representations | 127 |
| 7.1.1. Using Lists for Algebraic Arities & Signatures | 127 |
| 7.1.2. Modules and Morphisms for Arities | 128 |
| 7.2. Construction of the Initial Object | 133 |
| 7.2.1. The Terms Generated by a Signature | 133 |
| 7.2.2. Monad Structure on the Set of Terms | 134 |
| 7.2.3. A Representation in the Monad of Terms | 136 |
| 7.2.4. Weak Initiality for the Representation in the Term Monad | 137 |
| 7.2.5. Uniqueness and Initiality | 138 |
| 7.3. Remarks | 139 |
| 8. INITIALITY FOR UNTYPED 2-SIGNATURES, FORMALIZED | 141 |
| 8.1. Arities by Lists | 141 |
| 8.2. Representations of a 1-Signature | 142 |
| 8.3. Morphisms of Representations | 144 |
| 8.4. Category of Representations | 145 |
| 8.5. Initiality without Inequations | 146 |
| 8.6. Inequations and Initial Representation of a 2-Signature | 148 |
| 8.7. $\lambda\beta$: Lambda Calculus with beta reduction | 151 |
| 9. A FAITHFUL TRANSLATION OF PCF TO ULC | 155 |
| 9.1. Representations of PCF | 155 |
| 9.2. Morphisms of Representations | 158 |
| 9.3. Digression on Equal Fibre Modules in Coq | 160 |
| 9.4. Equality of Morphisms, Category of Representations | 162 |
| 9.5. One Particular Representation | 164 |
| 9.6. Initiality | 167 |
| 9.7. A Representation of PCF in the Untyped Lambda Calculus | 170 |
| 10. CONCLUSIONS AND FURTHER WORK | 175 |
| 10.1. Contributions | 175 |
| 10.2. Further Work | 175 |

Contents

| | |
|---|------------|
| A. SYNTAX AND SEMANTICS OF LAMBDA CALCULUS AND PCF | 179 |
| A.1. Syntax of Lambda Calculus and PCF | 179 |
| A.2. Semantics of Lambda Calculus and PCF | 180 |
| BIBLIOGRAPHY | 183 |

1. INTRODUCTION

In this thesis we give a characterization, via a universal property, of the syntax and semantics of simply-typed languages with variable binding. More precisely, we characterize the terms and sorts associated to a signature, equipped with reduction rules, as the initial object in some category. Via the iteration principle stemming from initiality, translations between languages, possibly over different sets of sorts, can be specified in a convenient and economic way. Furthermore, translations thus specified are ensured to be faithful with respect to reduction in the source and target languages, as well as compatible in a suitable sense with substitution on either side.

1.1. Motivation: Translations from PCF to ULC

As an introductory example, consider translations from the programming language PCF, introduced by Plotkin [Plo77], to the untyped lambda calculus ULC, invented by Church [Chu36]. A detailed account of both languages is given in Appx. A. These two languages are paradigmatic in the sense that PCF may be considered a rather high-level language, equipped with a type system, whereas the untyped lambda calculus represents a low-level, untyped language. We specify a map f from the set of PCF terms to the set of lambda terms as in Fig. 1.1 (cf. [Pho93]), with a suitable function g from the set of constants of PCF to lambda terms, e.g., $g(\mathbf{T}) := \lambda x y. x$, and suitable constants of the

$$\begin{aligned} f(\perp_A) &= \Omega \\ f(c_A) &= g(c) \\ f(x_A) &= x \\ f(s@t) &= f(s)@f(t) \\ f(\lambda x.M) &= \lambda x.f(M) \\ f(\mathbf{Fix}_A(M)) &= \Theta@f(M) \end{aligned}$$

Figure 1.1.: Translation from PCF to ULC

1. Introduction

lambda calculus, e.g.,

$$\begin{aligned}\Theta &:= (\lambda x. \lambda y. (y(xy))) (\lambda x. \lambda y. (y(xy))) \quad (\text{Turing fixed point combinator}) \text{ and} \\ \Omega &:= (\lambda x. xx)(\lambda x. xx) .\end{aligned}$$

Of course, different such translations exist; for instance, one may choose to translate **Fix** to a different fixed point combinator or one chooses a different representation g' for the constants of PCF in the lambda calculus, yielding a different translation $f' : \text{PCF} \rightarrow \text{ULC}$.

In this thesis we present a category-theoretic framework to specify such translations of a language to another. The challenges are

- the varying sets of sorts in source and target languages¹ and
- to capture compatibility of such translations with structure — such as substitution and reduction — in the source and target languages.

We construct a category in which “languages such as PCF and ULC are objects”, and in which the above translation $f : \text{PCF} \rightarrow \text{ULC}$ is a morphism. As it turns out, the preceding sentence is imprecise and needs to be refined: more precisely, in the category we construct the translation f is an *initial morphism* $f : \text{PCF} \rightarrow \text{ULC}$, that is, its source PCF is the initial object. Now, as we have seen, there are several possible translations from PCF to the lambda calculus, and the above translation $f : \text{PCF} \rightarrow \text{ULC}$ cannot be an initial morphism in a category where objects are “just” languages — otherwise we would have $f = f'$ for any translation $f' : \text{PCF} \rightarrow \text{ULC}$. Thus the objects in the category we construct are not just languages, but languages with additional structure, allowing us to distinguish different initial morphisms $f, f' : \text{PCF} \rightarrow \text{ULC}$,

$$\begin{array}{ccc} & & (\text{ULC}, \psi) \\ & \nearrow f & \\ (\text{PCF}, \phi) & & \\ & \searrow f' & \\ & & (\text{ULC}, \psi'). \end{array}$$

In this category, initiality of (PCF, ϕ) yields the following iteration principle: specifying an iterative translation $f : \text{PCF} \rightarrow \text{ULC}$ is equivalent to specifying the “extra structure” ψ of the lambda calculus ULC. We do not yet explain what this additional structure, here denoted ϕ, ψ and ψ' , looks like, but refer instead to [Sect. 1.2.1](#) for an instructive example.

A natural question then is whether — or better, in what sense — the translation f specified in [Fig. 1.1](#) is compatible with the respective reductions in the source and target

¹Here we consider untyped languages to be single-sorted.

languages. Phoa [Pho93] gives an answer to this question; in particular, the translation f is *faithful* in the sense that

$$t \rightarrow_{\text{PCF}} t' \text{ implies } f(t) \rightarrow_{\beta} f(t') .$$

In this thesis we provide a category-theoretic framework which allows to specify, via a universal property, such faithful translations between languages with variable binding over different sets of sorts.

1.2. Initial Semantics

Initial Semantics characterizes the terms of a language associated to a *signature* S as the initial object in some category — whose objects we call *Semantics of S* —, yielding a concise, high-level, definition of the abstract syntax associated to S . In more detail, the following “ingredients” are used:

Signature A *signature* specifies abstractly and concisely the syntax and semantics of a language.

Category of Representations To any signature S we associate a category of “models” of that signature, the objects of which we call *representations of S* .

Initiality In this category of representations of S we exhibit the initial object, the *language generated by S* .

The motivation for Initial Semantics are twofold: firstly, Initial Semantics provides a category-theoretic definition — via a universal property — of the syntax and semantics freely generated by a signature. Secondly, initiality yields an *iteration operator* which allows for an economic and convenient specification of morphisms — *translations* — from the initial object to other languages.

Depending on the “richness” of the language we want to define, we need a suitable notion of signature and, accordingly, of representation of that signature. The language features we consider in this thesis are the following:

Variable binding We consider binding constructors on the *term level*, such as lambda abstraction.

Typing We consider *simple* type systems, such as the simply-typed lambda calculus and, via the Curry–Howard isomorphism, propositional logic (cf. Sect. 3.4).

Reduction We consider semantics in form of reduction rules on terms, such as beta reduction,

$$\lambda x.M(N) \rightsquigarrow M[x := N] .$$

For the integration of each of the features above, the notions of signature and representation have to be adapted to accommodate the increasing amount of information which must be given to uniquely specify a language.

One of our goals is to use Initial Semantics in order to treat the last question of the preceding section: we would like to translate from one language into another — possibly over different sets of sorts —, using a universal, category-theoretic construction. This construction should take into account as much “structure” as possible. By this we mean that the translations under consideration should by construction be compatible, for instance, with typing and reduction in the source and target language. A more in-depth description of those structures is given in [Sects. 1.2.3, 1.2.4, 1.2.5 and 1.2.6](#).

In [Sect. 1.2.1](#) we explain the notion of *signature* and *representation* for a simple inductive data type, the natural numbers. The following sections sketch the changes that have to be made in order to integrate variable binding, substitution, typing and reduction rules, respectively. In [Sect. 1.3](#) we summarize the contributions of this thesis, whereas in [Sect. 1.4](#) we give a section-wise overview of its contents.

1.2.1. Example: Peano Axioms

We introduce the notion of *signature* and *representation* using the example of the natural numbers; in line with the triple structure mentioned at the beginning of [Sect. 1.2](#), our goal is to give a signature for the natural numbers and to associate to it a category of representations whose initial object is given by the natural numbers.

As a suitable *signature*, consider the following map from a two elements set to natural numbers:

$$\mathcal{N} := \{z \mapsto 0, \quad s \mapsto 1\}.$$

Intuitively, it says that the natural numbers are built from two *constructors*, namely a 0-ary operator (i.e. a constant), say, z , — the zero constant — and a unary operator, say, s — the successor function.

A *representation* of the signature \mathcal{N} is given by a triple (X, Z, S) of a set X together with a constant $Z \in X$ and a unary operation $S : X \rightarrow X$. A morphism to another such triple (X_0, Z_0, S_0) is a map $f : X \rightarrow X_0$ such that

$$f(Z) = Z_0 \quad \text{and} \quad f \circ S = S_0 \circ f. \quad (1.2.1)$$

This category has an *initial object* $(\mathbb{N}, \text{Zero}, \text{Succ})$ given by the natural numbers \mathbb{N} equipped with the constant $\text{Zero} = 0$ and the successor function $\text{Succ} : \mathbb{N} \rightarrow \mathbb{N}$.

Initiality of \mathbb{N} gives a way to specify *iterative* functions [Ven00] from \mathbb{N} to any set X by equipping X with a constant $Z \in X$ and a unary map $S : X \rightarrow X$, i.e. making the set X the carrier of an object $(X, Z, S) \in \mathcal{N}$. A different choice of $Z' \in X$ and $S' : X \rightarrow X$ yields a different iterative map $\mathbb{N} \rightarrow X$.

Put differently, reading [Disp. \(1.2.1\)](#) dynamically rather than statically, i.e. as a reduction from left to right rather than as equations, shows that functions on the initial object \mathbb{N} can be defined by *pattern matching*, where the right-hand side of the matching must obey a particular form.

1.1 Remark *Digression on Natural Numbers Object:* The very same definition is also used to define a *natural numbers object* in any category \mathcal{C} with a terminal object $\mathbf{1}$; just replace $Z \in X$ and $S : X \rightarrow X$ by morphisms $z : \mathbf{1} \rightarrow X$ and $s : X \rightarrow X$ in \mathcal{C} . More precisely, we call natural numbers object the triple $(\mathbb{N} : \mathcal{C}, \text{Zero} : \mathbf{1} \rightarrow \mathbb{N}, \text{Succ} : \mathbb{N} \rightarrow \mathbb{N})$ if, for any triple (X, z, s) of an object $X \in \mathcal{C}$ and morphisms z and s as above, there exists a unique morphism $f : \mathbb{N} \rightarrow X$ such that the following diagrams commute:

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\text{Zero}} & \mathbb{N} \\ & \searrow z & \downarrow f \\ & & X \end{array} \qquad \begin{array}{ccc} \mathbb{N} & \xrightarrow{\text{Succ}} & \mathbb{N} \\ f \downarrow & & \downarrow f \\ X & \xrightarrow{s} & X \end{array}$$

For details we refer to Mac Lane and Moerdijk's book [[MLM92](#)].

1.2.2. Initial Algebras

The term “Initial Algebra” is best explained using another viewpoint, where a signature is given by a *signature functor* $\Sigma : \text{Set} \rightarrow \text{Set}$. The category in question then is the category $\Sigma\text{-Alg}$ of algebras of the functor Σ , that is, the category whose objects are pairs (X, f) of a set X and a map $f : \Sigma X \rightarrow X$. A morphism to another such algebra (Y, g) is given by a map $h : X \rightarrow Y$ such that

$$\begin{array}{ccc} \Sigma X & \xrightarrow{\Sigma h} & \Sigma Y \\ f \downarrow & & \downarrow g \\ X & \xrightarrow{h} & Y \end{array}$$

commutes. The example of [Sect. 1.2.1](#) is equivalently given by the signature functor $\mathcal{N} : X \mapsto 1 + X$, with initial algebra

$$1 + \mathbb{N} \xrightarrow{[\text{Zero}, \text{Succ}]} \mathbb{N} .$$

Another example is that of *lists* (of finite length) of a given type A : let $F(X) := 1 + A \times X$. The initial F -algebra is given by the set $[A]$ of lists over A ,

$$1 + A \times [A] \xrightarrow{[\text{nil}, \text{cons}]} [A] .$$

1.2.3. Adding Variable Binding

When passing to syntax *with variable binding*, the question of how to model binding arises. The following representations of binding are among the most frequently used:

- Nominal syntax using named abstraction (\mathbb{A} being a set of *atoms*), e.g.,

$$\lambda : [\mathbb{A}]T \rightarrow T$$

- Higher-Order Abstract Syntax (HOAS), e.g.,

$$\lambda : (T \rightarrow T) \rightarrow T$$

and its *weak* variant, e.g.,

$$\lambda : (\mathbb{A} \rightarrow T) \rightarrow T$$

- Nested Data Types as presented in [BM98], e.g.,

$$\lambda : T(X + 1) \rightarrow T(X)$$

Note that the encoding via nested data types differs conceptually from the others in that here the set T of terms is parametrized explicitly by a *context*, i.e. a set X of variables possibly appearing freely in the terms of $T(X)$. Thus $T(X)$ denotes the set of terms of the language T with free variables in the set X . The set $X + 1$ corresponds to an *extended context* with one additional free variable, which is bound in the abstracted term. It is usually implemented through an inductive data type (`option` in OCAML or the `Maybe monad` in HASKELL) — whence the term “*Nested*”. It is also known under the name “Heterogenous data type” [AR99].

1.2 Example: We represent the untyped lambda calculus as a *nested data type* as done, e.g., by Bird and Paterson [BP99]: consider the following inductive type $\text{ULC} : \text{Set} \rightarrow \text{Set}$ of terms of the untyped lambda calculus²:

Inductive $\text{ULC} (V : \text{Type}) : \text{Type} :=$
 | $\text{Var} : V \rightarrow \text{ULC } V$
 | $\text{Abs} : \text{ULC } (\text{option } V) \rightarrow \text{ULC } V$
 | $\text{App} : \text{ULC } V \rightarrow \text{ULC } V \rightarrow \text{ULC } V$.

For syntax with binding, arities need to carry information about the binding behaviour of their associated constructor. One way to define such arities is using lists of natural numbers. The length of a list then indicates the number of arguments of the constructor,

²We use “Set” synonymously to “Type”. Note however, that types behave differently from sets in some aspects. In particular, given two (propositionally) equal types $A = B$ and $a : A$, we do not have $a : B$.

and the i -th entry denotes the number of variables that the constructor binds in the i -th argument. Continuing [Ex. 1.2](#), the signature Λ of ULC is given by

$$\Lambda := \{\text{app} : [0, 0] \ , \ \text{abs} : [1]\} \ .$$

The map $V \mapsto \text{ULC}(V)$ is in fact functorial: given a map $f : V \rightarrow W$, the map $\text{ULC}(f) : \text{ULC}(V) \rightarrow \text{ULC}(W)$ *renames* any free variable $v \in V$ in a term by $f(v)$, yielding a term with free variables in W . Accordingly, the signature Λ should be represented in functors $F : \text{Set} \rightarrow \text{Set}$ instead of in sets, and natural transformations take the place of maps.

1.2.4. Adding Substitution

As mentioned at the beginning of [Sect. 1.2](#), we would like to integrate as much structure as possible into our category of “models”. One such structure is (*capture-avoiding*) *substitution* of free variables. To account for substitution, we consider not plain functors $F : \text{Set} \rightarrow \text{Set}$ as in the preceding paragraph, but instead *monads* on the category Set of sets. Monads are functors equipped with some extra structure, which we explain by the example of the untyped lambda calculus. The map $V \mapsto \text{ULC}(V)$ comes with a (*capture-avoiding*) simultaneous substitution operation: let V and W be two sets (of variables) and f be a map $f : V \rightarrow \text{ULC}(W)$. Given a lambda term $t \in \text{ULC}(V)$, we can replace each free variable $v \in V$ in t by its image under f , yielding a term $t' \in \text{ULC}(W)$. Furthermore we consider the constructor Var_V as a “variable-as-term” map, indexed by a set of variables V ,

$$\text{Var}_V : V \rightarrow \text{ULC}(V) \ .$$

Altenkirch and Reus [[AR99](#)] observed that the well-known algebraic structure of monad captures those two operations and their properties: substitution and variable-as-term map turn ULC into a monad ([Def. 2.65](#)) on the category of sets.

The monad structure of ULC should be compatible in a suitable sense with the constructors Abs and App of ULC: *substitution distributes over constructors*. To capture this distributivity, Hirschowitz and Maggesi [[HM07a](#)] consider *modules over a monad* (cf. [Def. 2.43](#)) — which generalize monadic substitution —, and morphisms of modules — which are natural transformations that are compatible with the module substitution in a suitable sense. Indeed, the maps

$$\begin{aligned} \text{ULC} &: V \mapsto \text{ULC}(V) \ , \\ \text{ULC}' &: V \mapsto \text{ULC}(V + 1) \text{ and} \\ \text{ULC} \times \text{ULC} &: V \mapsto \text{ULC}(V) \times \text{ULC}(V) \end{aligned}$$

are the underlying maps of such modules (cf. [Exs. 2.45, 2.46](#)), and the constructors Abs and App are morphisms of modules (cf. [Exs. 2.47, 2.74](#)).

1.2.5. Adding Types

Type systems exist with varying features, ranging from simply-typed syntax to syntax with dependent types, kinds, polymorphism, etc. By simply-typed syntax we mean a non-polymorphic syntax where the set of types is independent from the set of terms, i.e. type constructors only take types as arguments. In more sophisticated type systems, types may depend on terms, leading to more complex definitions of arities and signatures. The present work is only concerned with simply-typed languages, such as the simply-typed λ -calculus and PCF. We refer to the underlying set of types of a language as *object types* or *sorts*.

The goal of typing is to *classify* terms according to some criteria. As an example, one may ask whether a term is of function type, that is, whether it would make sense to apply it to another term. Once such a classification of terms is achieved, one can use typing information to *filter* terms according to their types, in order to pick out only those terms that have the desired type. The classification of terms through typing thus has a semantic flavour. However, we still subsume typing under the *syntactic* aspect, since it has an impact on the set of terms of the language.

One way to add types would be to make them part of the terms, as in “ $\lambda x : \mathbb{N}.x + 4$ ”. However, for *simple type systems* it is possible to separate the worlds of types and terms and consider typing as a map from terms to types, thus giving a simple mathematical structure to typing. How can we be sure that our terms are well-typed? Despite the separation of types and terms we still want typing to be tightly integrated into the process of building terms, in order to avoid constructing ill-typed terms. Separation of terms and types seems to contradict this goal. The answer lies in considering not *one* set of terms with a “typing map” to the set, say, T , of types, but a *family of sets*, indexed by the set T of object types. Term constructors then can be “picky” about what terms they take as arguments, accepting only those terms that have the suitable type. We also consider free variables to be equipped with an object type. Put differently, we do not consider terms over *one* set of variables, but over a family of sets of variables, indexed by the set of object types. In other words, we consider a context to be given by a family $(V_t)_{t \in T}$ of sets of variables, where $V_t := V(t)$ is the set of variables of object type t . We illustrate our point of view by means of the example of the simply-typed lambda calculus TLC:

1.3 Example: Let

$$T_{\text{TLC}} ::= * \mid T_{\text{TLC}} \rightsquigarrow T_{\text{TLC}}$$

be the set of types of the simply-typed lambda calculus. The set family of simply-typed lambda terms with free variables in V is given by the following inductive family:

Inductive TLC $(V : T \rightarrow \text{Type}) : T \rightarrow \text{Type} :=$
 | Var : forall t, V t \rightarrow TLC V t
 | Abs : forall s t TLC (V + s) t \rightarrow TLC V (s \rightsquigarrow t)
 | App : forall s t, TLC V (s \rightsquigarrow t) \rightarrow TLC V s \rightarrow TLC V t.

where $V + s := V + \{s\}$ denotes context extension by a variable of type $s \in T_{\text{TLC}}$ — the variable which is bound by the constructor $\text{Abs}(s, t)$. The variables s and t range over the set T_{TLC} of types. The signature describing the simply-typed lambda calculus is given in Exs. 3.23 and 3.47. The preceding paragraph about monads and modules applies to the simply-typed lambda calculus when replacing sets by families of sets indexed by T_{TLC} : the simply-typed lambda calculus can be given the structure of a monad (cf. Ex. 2.37)

$$\text{TLC} : \text{Set}^{T_{\text{TLC}}} \rightarrow \text{Set}^{T_{\text{TLC}}} .$$

The constructors of TLC are morphisms of modules (cf. Exs. 2.61, 2.56, 2.60).

This method of defining exactly the well-typed terms by organizing them into a family of sets parametrized by object types is called *intrinsic typing* [BHKM11] — as opposed to the *extrinsic typing*, where first a set of *raw* terms is defined, which is then filtered via a typing predicate. Intrinsic typing delegates object level typing to the meta language type system, such as the Coq type system in Ex. 1.3. In this way, the meta level type checker (e.g. Coq) sorts out ill-typed terms automatically: writing such a term yields a type error on the meta level.

Furthermore, the intrinsic encoding comes with a much more convenient recursion principle; a map to any other type system can simply be defined by specifying its image on the well-typed terms. When using extrinsic typing, a map on terms would either have to be defined on the set of *raw* terms, including ill-typed ones, or on just the well-typed terms by specifying an additional propositional argument expressing the welltypedness of the term argument. Benton et al. give detailed explanation about intrinsic typing in a recently published paper [BHKM11].

1.2.6. Adding Reductions

The *semantics* of a programming language describes how programmes of that language *evaluate*. For functional programming languages as considered in this thesis, evaluation — or *computation* — is done by *reduction*. As an example, the evaluation of the term $7 + 5$ of a hypothetical arithmetic programming language to its “value” 12 is done by a series of reductions, whose precise form depends on the semantics of the language in question. Typical *rules*, which specify how terms reduce, are given in Sect. A.2 for the example languages of the lambda calculus and PCF.

Given a set A of such reduction rules, one may consider the relation generated by these rules. More precisely, following Barendregt and Barendsen [BB94], we consider several *closures* of those rules:

Propagation into subterms A relation R is called *compatible* if it is closed under propagation into subterms, that is, if for any constructor f of arity n and any $i \leq n$,

$$M \rightsquigarrow_R N \Rightarrow f(x_1, \dots, x_{i_1}, M, x_{i+1}, \dots, x_n) \rightsquigarrow_R f(x_1, \dots, x_{i_1}, N, x_{i+1}, \dots, x_n) .$$

1. Introduction

Reduction A relation R is a *reduction relation* if it is compatible, reflexive and transitive.

Equivalence A relation R is a *congruence* if it is a compatible equivalence relation.

To the set A of rules we associate three relations generated by A , which are the smallest relations that contain A and are a compatible relation, a reduction relation and a congruence, respectively. We denote these relations, in this order, by \rightarrow_A , \twoheadrightarrow_A and $=_A$, respectively.

1.4 Remark *Digression on Reduction Strategies:* Suppose we have a term in which reduction rules are applicable in several places, such as in the term

$$((\lambda x.M)N)((\lambda y.M')N') ,$$

which is β -reducible in the operator and in the operand. Here the natural question arises where one should reduce at first, in the operator or in the operand (or both in parallel) — the question about the *reduction strategy*. More precisely, one considers the following two properties of rewrite systems:

Termination Are there infinite — non-terminating — chains of reductions?

Confluence Suppose a term t reduces both to t' as well as to t'' via two different reductions. Is there a term t''' such that both t' and t'' reduce to t''' ?

Termination and confluence together yield (strong) normalization, an important property of rewriting systems: in a strongly normalizing rewriting system, any reduction strategy yields the same value for a given term — in particular, any reduction strategy arrives at a value, i.e. at a term without any more reducible subterms. To illustrate the concept of termination, we give an example of a lambda term such that one reduction strategy terminates whereas another one does not; consider the term $(\lambda x.y)(\Omega\Omega)$ with $\Omega = (\lambda x.xx)$ and a free variable y . Reducing the outermost beta redex results in an irreducible term y in one step, whereas the strategy of reducing at first the operand $(\Omega\Omega)$ leads to an infinite chain of reductions.

In this thesis we are interested in the *reduction relation* generated by a set of rules. It differs from the congruence by the absence of a *symmetry* rule, which, while adequate for mathematical reasoning, yields a relation that is too coarse from a point of view of *computation*. In the words of Girard [GTL89], while the congruence generated by A emphasizes the *static* point of view of mathematics, the reduction relation associated to A emphasizes the *dynamic* point of view of computation.

To account for reductions, we consider functors and monads whose codomain is not the category of (families of) sets, but of (families of) *preordered sets*. The definition of monad requires the underlying functor to be an endofunctor, but we do not want to consider preordered contexts — what would be the meaning of this preorder? The restriction to

endofunctors was lifted by Altenkirch et al. [ACU10] through the introduction of *relative monads*. A relative monad is given by a functor — not necessarily endo — together with two operations very similar to monadic variables-as-terms and substitution. We thus consider, e.g., the lambda calculus, as a relative monad associating to any set X of variables a *preordered set of lambda terms* $(\text{ULC}(X), \rightarrow_\beta)$, where the preorder on $\text{ULC}(X)$ is given by the reduction relation \rightarrow_β generated by the beta rule of [Disp. \(A.2.1\)](#), cf. [Ex. 2.85](#).

1.3. Contributions

In this thesis we give, via a universal property, an algebraic characterization of simply-typed syntax equipped with semantics in form of reduction rules. More precisely, given a pair of a *signature* — specifying the types and terms of a language — and *inequations* over this signature — specifying reduction rules —, we characterize the terms of the language associated to this signature, equipped with reduction rules according to the given inequations, as the initial object of a category of “models”.

Our starting point is work on initiality for untyped syntax done by Hirschowitz and Maggesi [HM07a], and on its generalization to simply-typed syntax by Zsidó [Zsi10]. In a first step we extend Zsidó’s theorem [Zsi10, Chap. 6] to account for varying sorts, cf. [Sect. 1.3.1](#). Afterwards, we integrate reduction rules into Hirschowitz and Maggesi’s [HM07a] purely syntactic initiality result, cf. [Sect. 1.3.2](#). Finally we obtain our main theorem, which accounts for varying object types as well as reduction rules, by combining the aforementioned two results, cf. [Sect. 1.3.3](#).

Furthermore, for the untyped case (cf. [Sect. 1.3.2](#)), we provide a formalized proof in the proof assistant Coq of our result, yielding a machinery which, when fed with a signature for terms and a set of inequations, produces the abstract syntax associated to the signature, together with the reduction relation generated by the given inequations. For the simply-typed case, we formalize the instantiation of our main result (cf. [Sect. 1.3.3](#)) to the signature of the programming language PCF [Plo77].

We now explain our contributions and approaches in more detail:

1.3.1. Extended Initiality for Varying Sorts

In her PhD thesis [Zsi10, Chap. 6], Zsidó proves an initiality theorem for the abstract syntax associated to a simply-typed signature. However, the “models” (or representations) she considers, among which the abstract syntax is the initial one, are all models over the same set of sorts. In this way, the iteration principle obtained by initiality does not allow the specification of a translation to a term language over a different set of sorts. We adapt Zsidó’s theorem by introducing *typed signatures*. A typed signature (S, Σ) specifies a set of *sorts* via an algebraic signature S , as well as a set of simply-typed *terms* over

these sorts via a term signature Σ over S . A representation R of such a typed signature is then given by a representation of its signature S for sorts in some set $T = T_R$ as well as a representation of Σ in a monad — also called R — over the category Set^T . A morphism of representations $P \rightarrow R$ consists of a morphism f of the underlying representations of S , together with a morphism of representations of Σ , that is compatible in a suitable sense with the “translation of sorts” f . We show that the category of representations of (S, Σ) thus defined has an initial object, which integrates the sorts freely generated by S and the terms freely generated by Σ , typed over the sorts of S . Our definition of morphisms ensures that, for any translation specified via the iteration principle, the translation of terms is compatible with the translation of sorts with respect to the typing in the source and target languages.

To summarize, compared to Zsidó’s theorem [Zsi10, Chap. 6] we consider representations of a signature for terms over *varying* sets of sorts. However, since we specify the set of sorts via a signature S and thus implement the variation of sorts through morphisms of representations of S , our “initial set of sorts” necessarily has *inductive* structure.

1.3.2. Integrating Reduction Rules

In order to integrate reduction rules into our initiality results, we define a notion of *2-signature*. A 2-signature (Σ, A) is given by a (1-)signature Σ which specifies the terms of a language, and a set A of *inequations* over Σ . Intuitively, each inequation specifies a reduction rule, for instance the beta rule.

The *models* — or *representations* — of such a 2-signature are built from *relative monads* and *modules over relative monads*: given a 1-signature Σ , we define a representation of Σ to be given by a relative monad on the appropriate functor $\Delta : \text{Set} \rightarrow \text{Pre}$ (cf. Def. 2.13) together with a suitable morphism of modules (over relative monads) for each arity of Σ . Given a set A of inequations over Σ , we define a satisfaction predicate for the models of Σ ; we call *representation of (Σ, A)* each representation of Σ that satisfies each inequation of A . This predicate specifies a full subcategory of the category of representations of Σ . We call this subcategory the *category of representations of (Σ, A)* . We prove that this category has an initial object, which is built by equipping the initial representation of Σ — given by the terms freely generated by Σ — with a suitable reduction relation generated by the inequations of A .

With this initiality theorem for (Σ, A) we obtain a new iteration principle, and any translation specified via this principle is, by construction, compatible with the reduction relation in the source and target languages.

1.3.3. Main Theorem: Initiality for Simply-Typed Syntax with Reduction

Finally, we combine the above two theorems in order to obtain an initiality result which accounts for the motivating example of Sect. 1.1. More precisely, we define a 2-signature to be given by a typed signature (S, Σ) as in Sect. 1.3.1 together a set A of (S, Σ) -inequations analogous to Sect. 1.3.2, specifying reduction rules.

We define a category of representations of $((S, \Sigma), A)$ and prove that this category has an initial object. This initial representation integrates the types and terms freely generated by (S, Σ) , the terms being equipped with the reduction relation generated by the inequations of A .

1.3.4. A Computer Implementation for Specifying Syntax and Semantics

Above theorems are really meant to be implemented in a proof assistant. Such an implementation allows the specification of syntax and reduction rules via 2-signatures, yielding a highly automated mechanism to produce syntax together with certified substitution and iteration principle.

We prove the initiality theorem described in Sect. 1.3.2 in the proof assistant Coq [Coq10]. As an illustration we describe how to obtain the untyped lambda calculus with beta reduction via initiality.

Furthermore we formalize an instance of the theorem explained in Sect. 1.3.3, also in Coq. More precisely, we define the category of representations of the typed signature of PCF with inequations and prove that this category has an initial object. Afterwards, we give a representation of this signature in the relative monad ULC_β of the untyped lambda calculus with beta reduction, yielding a translation from PCF to ULC. Instructions on how to obtain the complete source code of our Coq library are available on

<http://math.unice.fr/laboratoire/logiciels>.

1.4. Synopsis

This thesis consists of two parts: Part I (Chapts. 2 to 5) describes and proves informally the theorems which constitute this thesis, whereas Part II (Chapts. 6 to 9) describes their implementation and verification in the proof assistant Coq [Coq10].

Chapter 2: Category-Theoretic Constructions. We recall the notions of monad and module over a monad, together with some important constructions of modules.

1. Introduction

Afterwards we state equivalent definitions of monads, modules and their morphisms in the style of Manes, emphasizing their *substitution* structure.

Then we recall Altenkirch et al.’s definition of *relative monads* and define suitable morphisms for such monads.

Finally we define *modules over relative monads* and show that the constructions of modules over monads carry over to modules over relative monads.

Chapter 3: Simple Type Systems. We present two initiality theorems for simple type systems:

In [Sect. 3.2](#) we present Zsidó’s initiality theorem [[Zsi10](#), Chap. 6]: it characterizes the syntax associated to a simply-typed signature S over a set T of object types as the initial object in a category of representations of S .

In [Sect. 3.3](#) we prove a variant of Zsidó’s theorem which allows for representations of a term signature over varying sets of sorts. We introduce the notion of *typed signature* in order to account for translations of sorts. A typed signature (S, Σ) is a pair consisting of a first-order *algebraic signature* S for sorts, and a higher-order signature Σ for terms over those sorts. A representation of a typed signature (S, Σ) is again a pair given by a representation of the sort signature S in a set T and a representation of the term signature Σ in a monad P over the category Set^T . We show that the category of representations of a typed signature has an initial object.

Finally, as an example, we use the iteration principle stemming from initiality in order to specify a *double negation translation* from classical to intuitionistic propositional logic, viewing propositions as types via the Curry–Howard isomorphism.

Chapter 4: Reductions for Untyped Syntax. We prove an initiality theorem for untyped languages with variable binding, equipped with reduction rules.

For the specification of such languages, we define a notion of *2-signature*, i.e. a signature consisting of two levels: a *syntactic* level — called *1-signature* —, which specifies the terms of the language, and a *semantic* level, which specifies reduction rules for those terms through *inequations*. A representation of such a *2-signature* (Σ, A) is any representation of the underlying 1-signature Σ which satisfies each inequation of A .

We define the category of representations of (Σ, A) as the full subcategory of representations of Σ whose objects satisfy the inequations of A . We prove that this subcategory has an initial object, integrating the terms generated by Σ and the reduction relation generated by the rules of A .

As a running example we consider the 2-signature of the untyped lambda calculus with beta reduction.

The implementation of the theorem in Coq is explained in [Chapt. 8](#).

Chapter 5: Simple Type Systems with Reductions. We prove the main result of this thesis: we generalize the initiality result from the preceding [Chapt. 4](#) to *simply-typed* syntax with reduction rules, in a way that allows for change of object types as in [Sect. 3.3](#).

More precisely, we generalize the definition of *2-signature* to allow for the underlying 1-signature to specify a simple type system as in [Sect. 3.3](#). Accordingly, the definition of inequation is extended to allow for the specification of reduction rules on such simple type systems. The main theorem of this chapter states that the category of representations of such a 2-signature has an initial object. This initial representation integrates the types and terms specified by the underlying 1-signature, and is equipped with the reduction relation generated by the inequations of the 2-signature.

Chapter 6: Formalizing Category Theory in Coq. This chapter serves as an introduction to the proof assistant Coq in general and our library of category theory used in the following chapters in particular. We describe the formalization of basic concepts such as categories, (relative) monads and modules over (relative) monads. In the course of the chapter we also describe some of the features of Coq that we use, such as *implicit* arguments, the [Program](#) framework and coercions.

Chapter 7: Formalization of Zsidó’s theorem. Building up on the library presented in [Chapt. 6](#), we describe the formalization of Zsidó’s initiality theorem from [Sect. 3.2](#) in Coq. At first we define a Coq data type of simply-typed signatures over a given object type T . Afterwards we associate a category of representations to any such signature and prove that this category has an initial object.

Chapter 8: Initiality for Untyped 2-Signatures, Formalized. We describe the implementation in Coq of the theorem proved informally in [Chapt. 4](#): the category of representations of a 2-signature has an initial object. The formal proof follows the informal proof very closely; the only noteworthy difference is that the initial object of the underlying 1-signature is constructed directly rather than through the adjunction proved in [Chapt. 4](#).

Finally we demonstrate how to specify the untyped lambda calculus with beta reduction through a 2-signature in our implementation.

Chapter 9: A Faithful Translation of PCF to ULC. We formalize in Coq an instance of the main theorem of the thesis (cf. [Chapt. 5](#)), for the 2-signature of PCF, equipped with reduction rules as presented in [Fig. A.4](#). In particular, we explain where we encounter difficulties when using intrinsic typing in an intensional type system.

By representing the signature of PCF in the monad of the untyped lambda calculus, we obtain a translation from PCF to ULC that is compatible with reductions in the source and target languages.

1.5. Related Work

In this section we review related work, in particular in the field of *Initial Semantics* (cf. [Sect. 1.5.2](#)), i.e. algebraic characterization of syntax (and their semantics) and in the field of formalization of syntax in proof assistants, cf. [Sect. 1.5.3](#).

1.5.1. Translations from PCF

Our main example is given by the programming language PCF, introduced by Plotkin [[Plo77](#)]. This language and its various semantics have been studied extensively. The following work is not concerned with algebraic characterization of programming languages, and thus not directly related to this thesis; it rather answers questions that we do not (yet) consider in our categorical setting:

Phoa [[Pho93](#)] studies the semantic aspect of a specific translation of PCF to the untyped lambda calculus, i.e. the behaviour of this translation and its compatibility with respect to reduction in the source and target language. The translation he considers is also the one we specify via initiality in [Chapt. 9](#). The main result of this work is that this translation is *adequate* in the sense that a PCF programme reduces to a natural number constant n of PCF if and only if its translation into the lambda calculus reduces to the corresponding church numeral c_n .

Riecke [[Rie93](#)] studies translations from PCF into itself, where source and target are equipped with different *reduction strategies* (cf. [Rem. 1.4](#)). We do not consider reduction strategies in this thesis.

1.5.2. Initial Semantics

We classify work in Initial Semantics according to the features it covers. We are interested, in no particular order, in the following features:

- Typing
- Variable binding
- Semantics through (in)equations

Initial Semantics for untyped syntax without variable binding is a result by Birkhoff [[Bir35](#)]. Goguen et al. [[GTWW77](#)] give an overview of the literature about initial algebra and spell out explicitly the connection between initial algebras and abstract syntax. In fact, Goguen et al. also treat the example of a programming language with variable binding, which they call “Simple Applicative Language” (SAL). However, they circumvent the algebraic treatment of variable binding by modelling binding through a family of unary constructors $\text{abs}_x : \text{exp} \rightarrow \text{exp}$ where x varies over a fixed set of variables.

1.5.2.1. Variable binding

When looking for an algebraic treatment of variable binding, the question of how to model binding arises. Some possible encodings have already been mentioned in Sect. 1.2.3, we repeat the list — in no particular order — for reasons of convenience:

1. Nominal syntax using *atom* abstraction:

$$\lambda : [\mathbb{A}]T \rightarrow T$$

2. Higher-Order Abstract Syntax (HOAS):

$$\lambda : (T \rightarrow T) \rightarrow T$$

and its *weak* variant:

$$\lambda : (\mathbb{A} \rightarrow T) \rightarrow T$$

3. Nested Data Types:

$$\lambda : T(X + 1) \rightarrow T(X)$$

In the following, the numbers in parentheses indicate the technique used for modelling variable binding in the respective work, according to the list given above. Initial Semantics for untyped syntax was presented by Gabbay and Pitts [GP99, (1)], Hofmann [Hof99, (2)], Fiore et al. [FPT99, (3)] and Hirschowitz and Maggesi [HM07a, (3)].

While Gabbay and Pitts work in a *set theory* enriched with *atoms* — which serve as object level variables —, Hofmann, Fiore et al. and Hirschowitz and Maggesi use *category-theoretic* notions to formalize syntax. The nominal approach initiated by Gabbay and Pitts is the only one among those mentioned that allows for a study of alpha conversion. For all others the notion of alpha convertibility and syntactic equality coincide.

Fiore et al.’s approach is based on the notion of signature functor and Σ -monoid, where the central concept of substitution is expressed in terms of strengths. Hirschowitz and Maggesi model substitution through monads, following Altenkirch and Reus’ (cf. [AR99]) characterization of the untyped lambda calculus as a monad on the category of sets. The connection between those two approaches is made precise in Zsidó’s PhD thesis [Zsi10] in form of adjunctions between the respective categories of models.

Later Gabbay and Hofmann [GH08] exhibit the relation between nominal techniques and presheaves, showing that through the nominal approach one considers in fact presheaves F that preserve pullbacks of monomorphisms, i.e. presheaves that are stable under intersection, $F(X \cap Y) = FX \cap FY$.

Fiore et al.’s approach was extended by Fiore [Fio02] to the simply-typed lambda calculus, and for general simply-typed syntax by Miculan and Scagnetto [MS03, (2)]. Both use an encoding of binding via nested data types. The relation to Higher-Order Abstract Syntax — as “terms with holes” — is made precise in the latter work [MS03,

1. Introduction

Proposition 1]. Hirschowitz and Maggesi’s approach was generalized to simply-typed syntax in Zsidó’s thesis [Zsi10]. It was also generalized to account for more general term formers such as explicit flattening $\mu : T \circ T \rightarrow T$ [HM12].

Some of the mentioned lines of work have been extended to integrate *semantic aspects* in form of reduction relations on terms into initiality results:

1.5.2.2. Incorporating Semantics

Ghani and L  th [GL03] present rewriting for algebraic theories without variable binding; they characterize equational theories (with a *symmetry* rule) resp. rewrite systems (with *reflexivity* and *transitivity* rule, but without *symmetry*) as *coequalizers* resp. *coinserter*s in a category of monads on the categories **Set** resp. **Pre**.

Fiore and Hur [FH07] have extended Fiore’s work to “second-order universal algebras”, thus integrating semantic aspects in form of *equations* into initiality results. In particular, Hur’s thesis [Hur10] is dedicated to *equational systems* for syntax with variable binding. In a “Further research” section [Hur10, Chap. 9.3], Hur suggests the use of preorders, or more generally, arbitrary relations to model *inequational* systems.

Hirschowitz and Maggesi [HM07a] prove initiality of the set of lambda terms modulo beta and eta conversion in a category of *exponential monads*. In an unpublished paper [HM07b] they introduce the notion of *half-equation* and *equation* — as a pair of parallel half-equations — that we adopt in this thesis. However, we reinterpret a pair of parallel half-equations as an *inequation* rather than as an equation. Accordingly, we use preorders to model semantic aspects of syntax. This emphasizes the *dynamic* viewpoint of reductions as *directed* equalities — or *rewrite rules* — rather than the *static*, mathematical viewpoint one obtains by considering symmetric relations.

However, we consider not (traditional) monads but instead *relative* monads — on the appropriate functor $\Delta : \mathbf{Set} \rightarrow \mathbf{Pre}$ (cf. Def. 2.13) — as defined by Altenkirch et al. [ACU10], that is, monads with different source and target categories: we consider *variables* as elements of unstructured sets, whereas the set of *terms* of a language carries structure in form of a reduction relation. In our approach variables and terms thus live in *different* categories, which is realized mathematically through the use of relative monads instead of regular monads.

T. Hirschowitz [Hir] defines a category **Sig** of 2-signatures for *simply-typed* syntax with reductions, and constructs an adjunction between **Sig** and the category **2CCat** of small cartesian closed 2-categories. He thus associates, to any 2-signature, a 2-category of types and terms satisfying a universal property. His approach differs from ours in the way in which variable binding is modelled: Hirschowitz encodes binding in a Higher-Order Abstract Syntax (HOAS) style through exponentials. Reduction relations are expressed by the existence of 2-cells.

1.5.3. Formalization of Syntax

The implementation and formalization of syntax has been studied by a variety of people. The POPLMARK challenge [ABF⁺05] is a benchmark which aims to evaluate readability and provability when using different techniques of variable binding. However, the benchmark only concerns *one specific* language, not arbitrary syntax specified by a signature. The technique we use, called *Nested Abstract Syntax*, is used in a partial solution by Hirschowitz and Maggesi [HM10b], but was proposed earlier by others, see e.g. [BM98, AR99]. The use of *intrinsic typing* by dependent types of the meta-language was advertised in [BHKM11].

During our work we became aware of Capretta and Felty’s framework for reasoning about programming languages [CF09]. They implement a tool — also in the Coq proof assistant — which, given a signature, provides the associated abstract syntax as a data type dependent on the object types, hence intrinsically typed as well. Their data type of terms does not, however, depend on the set of free variables of those terms. Variables are encoded with de Bruijn indices. There are two different constructors for free and bound variables which serve to control the binding behaviour of object level constructors. In our theorem, there is only one constructor for (free) variables, and binding a variable is done by removing it from the set of free variables. Capretta and Felty then add a layer to translate those terms into syntax using named abstraction, and provide suitable induction and recursion principles. However, they do not consider *semantic* aspects, such as reduction rules, in their work.

The tool Ott [SNO⁺10] allows the specification of syntax and reduction rules, even for polymorphic type systems, in a system-independent ASCII file with subsequent translation into several different formal systems, including Coq, Isabelle [Pau88] and others. However, no algebraic characterization of the produced syntax is given.

1.5.4. Published Work

This thesis is partly based on the following articles:

Initial Semantics for higher-order typed syntax in Coq (with J. Zsidó) [AZ11]

The content of this article corresponds to the contents of Sect. 3.2 and Chapt. 7.

Extended Initiality for Typed Abstract Syntax [Ahr12]

The content of this article corresponds to the contents of Sect. 3.3 and Sect. 3.4.

Modules over relative monads for syntax and semantics [Ahr11]

The content of this article corresponds to the contents of Chapt. 4 and Chapt. 8.

PART I.

THEORY

2. CATEGORY-THEORETIC CONSTRUCTIONS

In this chapter, we first present some basic category-theoretic definitions (cf. [Sect. 2.1](#)). Afterwards, we review two different definitions of monads and modules over monads (cf. [Sects. 2.2](#) and [2.3](#)). Finally, we present relative monads and define colax morphisms of relative monads as well as modules over relative monads (cf. [Sect. 2.4](#)).

2.1. Categories, Functors & Transformations

In order to fix notations, we state some basic definitions of category theory, in particular those of category, functor and natural transformation. The examples we give in this section are used in later chapters. The reader might want to skip this section — throughout the thesis we link back to the definitions and examples where necessary.

The present section is not meant to constitute an introduction to category theory, nor does it define *all* of the concepts we use in the course of this work. For both an introduction to category theory as well as a reference for notions whose definitions are not given in this thesis, we refer to Mac Lane’s book [[ML98](#)].

2.1.1. Two Definitions of Categories

2.1 Definition (Category, [Code 6.3](#)): A category \mathcal{C} is given by

- a class — which we will also call \mathcal{C} — of *objects*,
- for any two objects c and d of \mathcal{C} , a class of *morphisms*, written $\mathcal{C}(c, d)$,
- for any object c of \mathcal{C} , a morphism $\text{id}_c \in \mathcal{C}(c, c)$ and
- for any three objects c, d, e of \mathcal{C} , a *composition* operation

$$(_ \circ _)_{c,d,e} : \mathcal{C}(d, e) \times \mathcal{C}(c, d) \rightarrow \mathcal{C}(c, e)$$

such that the composition is associative and the morphisms of the form id_c for suitable objects c are left and right neutral with respect to this composition¹:

$$\begin{aligned} \forall a \ b \ c \ d : \mathcal{C}, \forall f : \mathcal{C}(a, b), g : \mathcal{C}(b, c), h : \mathcal{C}(d, e), \quad (h \circ g) \circ f &= h \circ (g \circ f) \\ \forall c \ d : \mathcal{C}, \forall f : \mathcal{C}(c, d), \quad \text{id}_d \circ f &= f \text{ and } f \circ \text{id}_c = f \end{aligned}$$

¹We omit the “object” parameters from the composition operation, since those are deducible from the morphisms we compose. This omission is done in our library as well, via *implicit arguments* (cf. [Sect. 6.2](#)).

2. Category–Theoretic Constructions

We also write $f : c \rightarrow d$ for a morphism $f \in \mathcal{C}(c, d)$.

2.2 Remark: We omit a fifth condition stating that the classes of morphisms are pointwise disjoint. This condition is automatically satisfied when implementing the morphisms of a category as a dependent type of an intensional type theory, which we do in [Chapt. 6](#).

2.3 Remark (*Equivalent Def. of Category*): Equivalently to [Def. 2.1](#), a category \mathcal{C} is given by

- a class \mathcal{C}_0 of objects and a class \mathcal{C}_1 of morphisms,
- two maps denoting the source and target object of any morphism,

$$\text{src}, \text{tgt} : \mathcal{C}_1 \rightarrow \mathcal{C}_0 ,$$

- a partially defined composition function

$$(_ \circ _) : \mathcal{C}_1 \times \mathcal{C}_1 \rightarrow \mathcal{C}_1 ,$$

such that $g \circ f$ is defined only for *composable morphisms* f and g , i.e. for morphisms f and g such that $\text{tgt}(f) = \text{src}(g)$ — in which case we require that $\text{src}(g \circ f) = \text{src}(f)$ and $\text{tgt}(g \circ f) = \text{tgt}(g)$ —,

- an identity morphism for each object, i.e. a map

$$\text{id} : \mathcal{C}_0 \rightarrow \mathcal{C}_1 ,$$

such that $\text{src}(\text{id}(c)) = \text{tgt}(\text{id}(c)) = c$ and

- properties analogous to those of the preceding definition. The associative law, e.g., reads as

$$\forall f \ g \ h : \mathcal{C}_1, \ \text{tgt}(f) = \text{src}(g) \implies \text{tgt}(g) = \text{src}(h) \implies (h \circ g) \circ f = h \circ (g \circ f) .$$

While the two definitions of categories of [Def. 2.1](#) and of [Rem. 2.3](#) are equivalent, they both have some advantages and inconveniences when implementing them in a dependent type theory such as Coq. We expand on these differences in [Sect. 6.3.1](#).

2.4 Definition: The category **Set** has sets as objects. Morphisms from a set A to a set B are the total maps from A to B , together with the usual composition of maps.

Given a category \mathcal{C} , a morphism $f : c \rightarrow d$ from object c to object d is called *invertible*, if there exists a left- and right-inverse $g : d \rightarrow c$, that is, a morphism $g : d \rightarrow c$ such that $g \circ f = \text{id}_c$ and $f \circ g = \text{id}_d$. In this case the objects c and d are called *isomorphic*.

The following *universal property* plays a central rôle in this thesis:

2.5 Definition: Let \mathcal{C} be a category. The object c of \mathcal{C} is called *initial* if there exists precisely *one* morphism $i_d : c \rightarrow d$ in \mathcal{C} to any object d of \mathcal{C} .

Any two initial objects of a category \mathcal{C} are canonically isomorphic. We usually do not distinguish canonically isomorphic objects of a category, which explains the (standard) use of the definite article. Whenever it exists, we also write $0_{\mathcal{C}}$ — or simply 0 , when the category in question can be deduced from the context — for the initial object of \mathcal{C} . The dual concept is that of a *terminal* object:

2.6 Definition: Let \mathcal{C} be a category. The object d of \mathcal{C} is called *terminal* if there exists precisely *one* morphism $t_c : c \rightarrow d$ in \mathcal{C} from any object c of \mathcal{C} .

2.7 Example: The empty set is initial in the category \mathbf{Set} of sets. The singleton set is terminal in \mathbf{Set} .

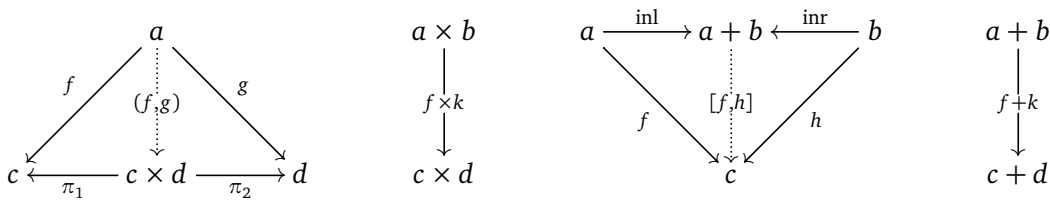
Later we also use the following categories:

2.8 Definition: The category \mathbf{Pre} of preorders has, as objects, sets equipped with a preorder, and, as morphisms between any two preorders A and B , the monotone functions from A to B .

2.9 Definition: The category \mathbf{wPre} has, as objects, sets equipped with a preorder, and, as morphisms between any two preordered sets A and B , all set-theoretic maps from A to B , not necessarily monotone.

2.10 Example: Any set T can be regarded as a *discrete* category, with objects the elements of T , and just identity morphisms.

2.11 Notation Product, Coproduct: We refer to Mac Lane’s book [ML98] for the definition of product and coproduct. Whenever they exist, we write $a \times b$ for the product of objects a and b of \mathcal{C} , and $a + b$ for the coproduct. Notation for arrows is informally explained in the following diagrams:



2.1.2. Functors & Natural Transformations

Given two categories \mathcal{C} and \mathcal{D} , a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ maps objects of \mathcal{C} to objects of \mathcal{D} , and morphisms of \mathcal{C} to morphisms of \mathcal{D} , while preserving source and target as well as composition and identity:

2. Category–Theoretic Constructions

2.12 Definition: A functor F from \mathcal{C} to \mathcal{D} is given by

- a map $F : \mathcal{C} \rightarrow \mathcal{D}$ on the objects of the categories involved and
- for any pair of objects (c, d) of \mathcal{C} , a map

$$F_{(c,d)} : \mathcal{C}(c, d) \rightarrow \mathcal{D}(Fc, Fd) ,$$

such that

- $\forall c : C, F(id_c) = id_{Fc}$ and
- $\forall c, d, e : C, \forall f : c \rightarrow d, \forall g : d \rightarrow e, F(g \circ f) = Fg \circ Ff$.

Here we use the same notation for the map on objects and that on morphisms. For the latter we also omit the subscript “ (c, d) ” as *implicit* arguments.

2.13 Definition (Functor $\Delta : \text{Set} \rightarrow \text{Pre}$ and Forgetful Functor): We call $\Delta : \text{Set} \rightarrow \text{Pre}$ the functor from sets to preordered sets which associates to each set X the set itself together with the smallest preorder, i.e. the diagonal of X ,

$$\Delta(X) := (X, \delta_X).$$

In other words, for any $x, y \in X$ we have $x\delta_X y$ if and only if $x = y$. The functor $\Delta : \text{Set} \rightarrow \text{Pre}$ is a *full embedding*, i.e. it is fully faithful and injective on objects.

In the other direction we have a *forgetful* functor $U : \text{Pre} \rightarrow \text{Set}$ which maps any preordered set (X, \leq) to the set X . We have $U \circ \Delta = \text{Id}_{\text{Set}}$.

2.14 Definition (Natural Transformation): Given two functors $F, G : \mathcal{C} \rightarrow \mathcal{D}$, a natural transformation $\gamma : F \rightarrow G$ (also written $\gamma : F \Rightarrow G$) is given by a family of morphisms

$$\gamma_c : \mathcal{D}(Fc, Gc)$$

indexed by objects of \mathcal{C} such that, for any morphism $f : c \rightarrow d$ in \mathcal{C} , the following diagram commutes:

$$\begin{array}{ccc} Fc & \xrightarrow{\gamma_c} & Gc \\ Ff \downarrow & & \downarrow Gf \\ Fd & \xrightarrow{\gamma_d} & Gd \end{array}$$

2.15 Definition (Adjunction): Let \mathcal{C} and \mathcal{D} be categories. An adjunction from \mathcal{C} to \mathcal{D} is given by

- a functor $F : \mathcal{C} \rightarrow \mathcal{D}$,
- a functor $G : \mathcal{D} \rightarrow \mathcal{C}$,
- a natural transformation $\eta : \text{Id}_{\mathcal{C}} \rightarrow G \circ F$, called *unit*, and

- a natural transformation $\epsilon : F \circ G \rightarrow \text{Id}_{\mathcal{D}}$, called *counit*,

such that the transformations

$$G \xrightarrow{\eta G} GFG \xrightarrow{G\epsilon} G, \quad F \xrightarrow{F\eta} FGF \xrightarrow{\epsilon F} F$$

both are the identity transformation. We write $F \dashv G$ for such an adjunction, leaving the unit and counit implicit.

2.16 Remark: The functors F and G as above are adjoint if and only there is a family of bijections

$$\varphi = (\varphi_{c,d} : \mathcal{D}(Fc, d) \cong \mathcal{C}(c, Gd))$$

indexed by objects $c, d \in \mathcal{C}$, which is natural in both c and d .

2.17 Definition (Coreflection): Let $F : \mathcal{C} \rightarrow \mathcal{D}$ be an embedding, that is, a faithful functor which is injective on objects — e.g., the inclusion of a subcategory. Then F is a *coreflection* if it has a right adjoint.

The following lemma gives an example of a coreflection:

2.18 Lemma: The forgetful functor $U : \text{Pre} \rightarrow \text{Set}$ is right adjoint to the diagonal functor $\Delta : \text{Set} \rightarrow \text{Pre}$:

$$\text{Set} \begin{array}{c} \xrightarrow{\Delta} \\ \perp \\ \xleftarrow{U} \end{array} \text{Pre} ,$$

that is, the embedding $\Delta : \text{Set} \rightarrow \text{Pre}$ is a coreflection. We denote by φ the family of isomorphisms

$$\varphi_{X,Y} : \text{Pre}(\Delta X, Y) \cong \text{Set}(X, UY) .$$

We omit the indices of φ whenever they can be deduced from the context.

Proof. The unit is given by a family of identity maps $\eta_X := \text{id}_X : \text{Set}(X, U\Delta X)$. The counit is given by a family of maps $\epsilon_Y : \text{Pre}(\Delta UY, Y)$ whose carrier map on UY is the identity map on UY . \square

We later use the following result about left adjoints:

2.19 Lemma (Left adjoints are cocontinuous): Left adjoints are cocontinuous, i.e. commute with colimits. In particular, the image of an initial object under a left adjoint is initial.

For the proof we refer to Mac Lane's book [ML98, V5.Thm.1].

2.1.3. More Examples, Notations

The following categories and functors will appear in different places throughout the thesis. Again, the reader may skip these examples for the moment; we will point to the definitions from the place where they are used.

2.20 Definition (Category of Families): Let \mathcal{C} be a category and T be a set, i.e. a discrete category (cf. [Ex. 2.10](#)). We denote by \mathcal{C}^T the functor category, an object of which is a T –indexed family of objects of \mathcal{C} . Given two families V and W , a morphism $f : V \rightarrow W$ is a family of morphisms in \mathcal{C} ,

$$f : t \mapsto f(t) : V(t) \rightarrow W(t) .$$

We write $V_t := V(t)$ for objects and morphisms. Given another category \mathcal{D} and a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, we denote by F^T the functor defined on objects and morphisms as

$$F^T : \mathcal{C}^T \rightarrow \mathcal{D}^T, \quad f \mapsto (t \mapsto F(f_t)) .$$

2.21 Remark: Given a set T , the adjunction of [Lem. 2.18](#) induces an adjunction

$$\text{Set}^T \begin{array}{c} \xrightarrow{\Delta^T} \\ \perp \\ \xleftarrow{U^T} \end{array} \text{Pre}^T .$$

2.22 Definition (Retyping Functor): Let T and T' be sets and $g : T \rightarrow T'$ be a map. Let \mathcal{C} be a cocomplete category. The map g induces a functor

$$g^* : \mathcal{C}^{T'} \rightarrow \mathcal{C}^T, \quad W \mapsto W \circ g .$$

The *retyping functor* associated to $g : T \rightarrow T'$,

$$\vec{g} : \mathcal{C}^T \rightarrow \mathcal{C}^{T'},$$

is defined as the left Kan extension operation along g , that is, we have an adjunction

$$\mathcal{C}^T \begin{array}{c} \xrightarrow{\vec{g}} \\ \perp \\ \xleftarrow{g^*} \end{array} \mathcal{C}^{T'} . \tag{2.1.1}$$

2.23 Remark *Retyping Functor Explicitly*, [Code 6.6](#): In the context of [Def. 2.22](#), we define the functor

$$\vec{g} : \mathcal{C}^T \rightarrow \mathcal{C}^{T'}, \quad X = t \mapsto X_t \mapsto \vec{g}(X) := t' \mapsto \coprod_{\{t \mid g(t)=t'\}} X_t .$$

In particular, for any $V \in \mathcal{C}^T$ — considered as a functor — we have a natural transformation

$$V \Rightarrow \bar{g}V \circ g : T \rightarrow \mathcal{C}$$

given pointwise by the morphism $V_t \rightarrow \coprod_{\{s \mid g(s)=g(t)\}} V_s$ in the category \mathcal{C} . Put differently, the map $g : T \rightarrow T'$ induces an endofunctor \bar{g} on \mathcal{C}^T with object map

$$\bar{g}(V) := \bar{g}(V) \circ g$$

and we have a natural transformation ctype — the unit of the adjunction of [Disp. \(2.1.1\)](#),

$$\text{ctype} : \text{Id} \Rightarrow \bar{g} : \mathcal{C}^T \rightarrow \mathcal{C}^T .$$

2.24 Remark: One can interpret the map $g : T \rightarrow T'$ as a translation of object sorts and the functor \bar{g} as a “retyping functor” which changes the sorts of contexts and terms (or more generally, models of terms) according to the translation of sorts. The monads we are interested in are monads over some category Set^T and our monad morphisms are over retyping functors. In [Chapt. 3](#) we interpret the syntax of a language P over a set of types T as a monad P over the category Set^T . Given another language Q over a set of types U , we consider a translation from P to Q to be a translation of object types $g : T \rightarrow U$ and a colax monad morphism $P \rightarrow Q$ over the retyping functor $\bar{g} : \text{Set}^T \rightarrow \text{Set}^U$ (cf. [Def. 2.38](#)).

2.25 Remark about maps on coproducts and pattern matching: In the proof assistant Coq we implement retyping (cf. [Rem. 2.23](#)) via an inductive family, cf. [Code 6.6](#). In this context, passing from the left to the right in the adjunction isomorphism

$$C^{T'}(\bar{g}V, W) \cong C^T(V, g^*W)$$

is done by precomposing with *pattern matching on the constructor* ctype , cf. [Sect. 9.6](#).

2.26 Definition (Pointed index sets): Given a category \mathcal{C} , a set T and a natural number n , we denote by \mathcal{C}_n^T the category with, as objects, diagrams of the form

$$n \xrightarrow{\mathbf{t}} T \xrightarrow{V} \mathcal{C} ,$$

written (V, t_1, \dots, t_n) with $t_i := \mathbf{t}(i)$. A morphism h to another such (W, \mathbf{t}) with the same pointing map \mathbf{t} is given by a morphism $h : V \rightarrow W$ in \mathcal{C}^T . Note that there are no morphisms between families with different points, that is, $\mathcal{C}_n^T((V, \mathbf{t}), (V', \mathbf{t}')) = \emptyset$ if $\mathbf{t} \neq \mathbf{t}'$. Any functor $F : \mathcal{C}^T \rightarrow \mathcal{D}^T$ extends to $F_n : \mathcal{C}_n^T \rightarrow \mathcal{D}_n^T$ via

$$F_n(V, t_1, \dots, t_n) := (FV, t_1, \dots, t_n) .$$

2.27 Remark: The category \mathcal{C}_n^T consists of T^n copies of \mathcal{C}^T , which do not interact. Due to the “markers” (t_1, \dots, t_n) we can act differently on each copy, cf., e.g., [Defs. 2.57](#) and [2.59](#). The reason why we consider categories of this form is explained at the beginning of [Sect. 3.3](#) and in [Rem. 3.37](#).

2. Category–Theoretic Constructions

Retyping functors generalize to categories with pointed indexing sets; when changing types according to a map of types $g : T \rightarrow U$, the markers must be adapted as well:

2.28 Definition: Given a map of sets $g : T \rightarrow U$, by postcomposing the pointing map with g , the retyping functor generalizes to the functor

$$\vec{g}(n) : \mathcal{C}_n^T \rightarrow \mathcal{C}_n^U, \quad (V, \mathbf{t}) \mapsto (\vec{g}V, g_*(\mathbf{t})),$$

where $g_*(\mathbf{t}) := g \circ \mathbf{t} : n \rightarrow U$.

Finally there is also a category where families of objects of \mathcal{C} over different indexing sets are mixed together:

2.29 Definition: Given a category \mathcal{C} , we denote by $\mathcal{T}\mathcal{C}$ the category where an object is a pair (T, V) of a set T and a family $V \in \mathcal{C}^T$ of objects of \mathcal{C} indexed by T . A morphism (g, h) to another such (T', W) is given by a map $g : T \rightarrow T'$ and a morphism $h : V \rightarrow W \circ g$ in \mathcal{C}^T , that is, a family of morphisms in \mathcal{C} , indexed by T ,

$$h_t : V_t \rightarrow W_{g(t)}.$$

Suppose \mathcal{C} has an initial object, denoted by $0_{\mathcal{C}}$. Given $n \in \mathbb{N}$, we call $\hat{n} = (n, k \mapsto 0_{\mathcal{C}})$ the object of $\mathcal{T}\mathcal{C}$ that associates to any $1 \leq k \leq n$ the initial object of \mathcal{C} . We call $\mathcal{T}\mathcal{C}_n$ the slice category $\hat{n} \downarrow \mathcal{T}\mathcal{C}$. An object of this category consists of an object $(T, V) \in \mathcal{T}\mathcal{C}$ whose indexing set “of types” T is pointed n times, written (T, V, \mathbf{t}) , where \mathbf{t} is a vector of elements of T of length n . A morphism $(g, h) : (T, V, \mathbf{t}) \rightarrow (T', V', \mathbf{t}')$ is a morphism $(g, h) : (T, V) \rightarrow (T', V')$ as above, such that $\mathbf{t}' = \mathbf{t} \circ g$.

We call $\mathcal{T}U_n : \mathcal{T}\mathcal{C}_n \rightarrow \mathbf{Set}$ the forgetful functor associating to any pointed family (T, V, t_1, \dots, t_n) the indexing set T . Note that for a fixed set T , the category \mathcal{C}_n^T (cf. [Def. 2.26](#)) is the fibre over T of this functor.

2.30 Remark Picking out Sorts: Let $1 : \mathcal{T}\mathcal{C}_n \rightarrow \mathbf{Set}$ denote the constant functor which maps objects to the terminal object of the category \mathbf{Set} . A natural transformation $\tau : 1 \rightarrow \mathcal{T}U_n$ associates to any object (T, V, \mathbf{t}) of the category $\mathcal{T}\mathcal{C}_n$ an element of T . Naturality imposes that $\tau(T', V', \mathbf{t}') = g(\tau(T, V, \mathbf{t}))$ for any $(g, h) : (T, V, \mathbf{t}) \rightarrow (T', V', \mathbf{t}')$.

2.31 Notation: Given a natural transformation $\tau : 1 \rightarrow \mathcal{T}U_n$ as in [Rem. 2.30](#), we write

$$\tau(T, V, \mathbf{t}) := \tau(T, V, \mathbf{t})(*) \in T,$$

i.e. we omit the argument $* \in 1_{\mathbf{Set}}$ of the singleton set.

2.32 Example: For $1 \leq k \leq n$, we denote by $k : 1 \Rightarrow \mathcal{T}U_n : \mathcal{T}\mathcal{C}_n \rightarrow \mathbf{Set}$ the natural transformation such that $k(T, V, \mathbf{t}) := \mathbf{t}(k)$.

2.2. Monads & Modules

We state the widely known definition of monad and the less known definition of *module over a monad*, together with their respective morphisms. Modules have been used in the context of Initial Semantics by Hirschowitz and Maggesi [HM07a, HM10a] and Zsidó [Zsi10]. The monad morphisms we are interested in are, more precisely, *colax monad morphisms*, see, e.g., Leinster’s book [Lei04].

2.2.1. Definitions

2.33 Definition (Monad): A *monad* T over a category \mathcal{C} is given by

- a functor $T : \mathcal{C} \rightarrow \mathcal{C}$ (which we denote by the same name as the monad),
- a natural transformation $\eta : \text{Id}_{\mathcal{C}} \rightarrow T$ and
- a natural transformation $\mu : T \circ T \rightarrow T$

such that the following diagrams commute:

$$\begin{array}{ccc}
 T & \xrightarrow{T\eta} & T^2 \\
 & \searrow \text{id} & \downarrow \mu \\
 & & T
 \end{array}
 \quad
 \begin{array}{ccc}
 T^3 & \xrightarrow{\mu_T} & T^2 \\
 T\mu \downarrow & & \downarrow \mu \\
 T^2 & \xrightarrow{\mu} & T
 \end{array}$$

2.34 Example (List Monad): The functor $[_] : \text{Set} \rightarrow \text{Set}$ which to any set X associates the set of finite lists over X , is equipped with a structure as monad by defining η and μ as “singleton list” and flattening, respectively:

$$\eta_X(x) := [x] \quad \text{and}$$

$$\mu_X \left([[x_{1,1}, \dots, x_{1,m_1}], \dots, [x_{n,1}, \dots, x_{n,m_n}]] \right) := [x_{1,1}, \dots, x_{1,m_1}, \dots, x_{n,1}, \dots, x_{n,m_n}].$$

2.35 Remark Kleisli Operation (Monadic Bind): Given a monad (T, η, μ) on the category \mathcal{C} , the Kleisli operation σ is defined, for any $a, b \in \mathcal{C}$ and $f \in \mathcal{C}(a, Tb)$, by setting

$$\begin{aligned}
 \sigma_{a,b} : \mathcal{C}(a, Tb) &\rightarrow \mathcal{C}(Ta, Tb) , \\
 f &\mapsto \mu_b \circ Tf .
 \end{aligned}$$

Indeed, a monad (T, η, μ) can equivalently be defined as a triple (T, η, σ) with an adapted set of axioms, see Def. 2.65. We often leave the object arguments a and b implicit, i.e. we write $\sigma(f) := \sigma_{a,b}(f)$.

2.36 Example (Monadic Syntax, untyped): Syntax as a monad (in form of a Kleisli triple) was presented by Altenkirch and Reus [AR99]: consider the syntax of the untyped lambda calculus ULC as given in Ex. 1.2 in Sect. 1.2.1. As mentioned there, the map $V \mapsto \text{ULC}(V)$ is functorial, its map on morphisms is given by *renaming* of free variables. This functor is equipped with a monad structure by defining η as variable–as–term operation

$$\eta_V(v) := \text{Var}(v) \in \text{ULC}(V)$$

and the multiplication $\mu : \text{ULC} \circ \text{ULC} \rightarrow \text{ULC}$ as flattening which, given a term of ULC with terms of $\text{ULC}(V)$ as variables, returns a term of $\text{ULC}(V)$ by removing a layer of intermediate Var constructors. These definitions turn (ULC, η, μ) into a monad on the category Set . The Kleisli operation associated to this monad corresponds to simultaneous substitution [AR99].

2.37 Example (Monadic Syntax, typed): Consider the syntax of the simply–typed lambda calculus as defined in Ex. 1.3. The map

$$\text{TLC} : \text{Set}^{T_{\text{TLC}}} \rightarrow \text{Set}^{T_{\text{TLC}}} , \quad V \mapsto \text{TLC}(V) ,$$

associating to any set family V the family of lambda terms with free variables in V , is the object map of a functor. Similarly to the untyped lambda calculus (cf. Ex. 2.36), the natural transformations $\eta : \text{Id} \rightarrow \text{TLC}$ and $\mu : \text{TLC} \circ \text{TLC} \rightarrow \text{TLC}$ are defined as variable–as–term operation and flattening, respectively. These definitions turn (TLC, η, μ) into a monad on the category $\text{Set}^{T_{\text{TLC}}}$.

Our definition of *colax* monad morphisms and their *transformations* is taken from Leinster’s book [Lei04]:

2.38 Definition (Colax Monad Morphism): Let (T, η, μ) be a monad on the category \mathcal{C} and (T', η', μ') be a monad on the category \mathcal{D} . A *colax morphism of monads* $(\mathcal{C}, T) \rightarrow (\mathcal{D}, T')$ is given by

- a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ and
- a natural transformation $\gamma : FT \rightarrow T'F$ as in

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{T} & \mathcal{C} \\ F \downarrow & \nearrow \gamma & \downarrow F \\ \mathcal{D} & \xrightarrow{T'} & \mathcal{D} \end{array}$$

such that the following diagrams commute:

$$\begin{array}{ccc}
 FTT & \xrightarrow{\gamma^T} & T'FT & \xrightarrow{\gamma} & T'T'F \\
 F\mu \downarrow & & & & \downarrow \mu'F \\
 FT & \xrightarrow{\gamma} & T'F, & &
 \end{array}
 \qquad
 \begin{array}{ccc}
 F & & \\
 F\eta \downarrow & \searrow \eta'F & \\
 FT & \xrightarrow{\gamma} & T'F.
 \end{array}$$

From now on we simply say “monad morphism over F ” when speaking about a colax monad morphism with underlying functor F .

2.39 Definition (Composition of Monad Morphisms): Suppose given a monad morphism as in Def. 2.38. Given a third monad (T'', η'', μ'') on category \mathcal{C} and a monad morphism $(F', \gamma') : (T', \eta', \mu') \rightarrow (T'', \eta'', \mu'')$, we define the composition of (F, γ) and (F', γ') to be the monad morphism given by the pair consisting of the functor $F'F$ and the transformation

$$F'FT \xrightarrow{F'\gamma} F'T'F \xrightarrow{\gamma'F} T''F'F .$$

The verification of the necessary commutativity properties is done — for the equivalent definition given in Def. 2.69 — in the Coq library, cf. `colax_Monad_Hom_comp`.

2.40 Definition (Transformation): Given two morphisms of monads

$$(F, \gamma), (F', \gamma') : (\mathcal{C}, T) \rightarrow (\mathcal{D}, T') ,$$

a *transformation* $(F, \gamma) \rightarrow (F', \gamma')$ is given by a natural transformation $\beta : F \Rightarrow F'$ such that the following diagram commutes:

$$\begin{array}{ccc}
 FT & \xrightarrow{\gamma} & T'F \\
 \beta T \downarrow & & \downarrow T'\beta \\
 F'T & \xrightarrow{\gamma'} & T'F'.
 \end{array}$$

A *2-category* is a category with “morphisms between morphisms”. We refer to Mac Lane’s book [ML98] for the definition.

2.41 Definition (2-Category of Monads, [Lei04]): We call $\mathbf{Mnd}_{\text{colax}}$ the 2-category an object of which is a pair (\mathcal{C}, T) of a category \mathcal{C} and a monad T on \mathcal{C} . A morphism to another object (\mathcal{D}, T') is a colax monad morphism $(F, \gamma) : (\mathcal{C}, T) \rightarrow (\mathcal{D}, T')$. A 2-cell $(F, \gamma) \Rightarrow (F', \gamma')$ is a transformation.

2.42 Notation: For any category \mathcal{C} , we write $\text{Id}_{\mathcal{C}}$ for the object (\mathcal{C}, Id) of $\mathbf{Mnd}_{\text{colax}}$.

We are interested in *modules over monads*. These are particular monad morphisms whose codomain is the identity monad on some category². Modules and, more specifically, their morphisms, capture the distributivity of substitution over the constructors of a language, cf. [Ex. 2.47](#) and [Ex. 2.74](#).

2.43 Definition (Module over a Monad): Given categories \mathcal{C} and \mathcal{D} and a monad T on \mathcal{C} , a *module over T with codomain \mathcal{D}* (or *T -module towards \mathcal{D}*) is a colax monad morphism $(M, \gamma) : (\mathcal{C}, T) \rightarrow (\mathcal{D}, \text{Id}_{\mathcal{D}})$ from T to the identity monad on \mathcal{D} . Given parallel T -modules M and N , a *morphism of modules from M to N* is a transformation from M to N as in [Def. 2.40](#). We denote the category of T -modules towards \mathcal{D} by

$$\text{Mod}(T, \mathcal{D}) := \mathbf{Mnd}_{\text{colax}}((\mathcal{C}, T), (\mathcal{D}, \text{Id})) .$$

Before giving some examples of modules over monads, we state a more explicit definition of modules:

2.44 Remark (*Modules and their Morphisms, explicitly* [[HMO7a](#)]): By unfolding the preceding definition and simplifying, we obtain that a T -module towards \mathcal{D} is a functor $M : \mathcal{C} \rightarrow \mathcal{D}$ together with a natural transformation $\sigma : MT \rightarrow M$ such that the following diagrams commute:

$$\begin{array}{ccc} MT T & \xrightarrow{\sigma T} & MT \\ M\mu \downarrow & & \downarrow \sigma \\ MT & \xrightarrow{\sigma} & M, \end{array} \quad \begin{array}{ccc} M & & \\ M\eta \downarrow & \searrow \text{id} & \\ MT & \xrightarrow{\sigma} & M. \end{array}$$

Such a module can hence be regarded as a kind of generalized monad over a functor that is not necessarily an endofunctor; indeed, this is our intuition behind modules. In particular, every monad gives rise to a module over itself, the *tautological module* (cf. [Def. 2.48](#)). Furthermore, the category of modules $\text{Mod}(T, \mathcal{D})$ allows for products, provided the target category \mathcal{D} is equipped with a product.

A morphism of T -modules from (M, σ) to (M', σ') then is given by a natural transformation $\beta : M \Rightarrow M'$ such that the following diagram commutes:

$$\begin{array}{ccc} MT & \xrightarrow{\beta T} & M' T \\ \sigma \downarrow & & \downarrow \sigma' \\ M & \xrightarrow{\beta} & M'. \end{array}$$

We anticipate the constructions of the next section by giving some examples of modules:

²The characterization of modules over monads as particular colax monad morphisms is due to an anonymous referee, whom I hereby thank for his helpful comments.

2.45 Example (Tautological Module, [Ex. 2.36](#) cont.): Any monad T on a category \mathcal{C} can be considered as a module over itself, the *tautological module* (cf. [Def. 2.48](#)). In particular, the monad of the untyped lambda calculus ULC (cf. [Ex. 2.36](#)) is a ULC-module with codomain Set .

2.46 Example: The map

$$\text{ULC}' : V \mapsto \text{ULC}(V') ,$$

with $V' := V + 1$, inherits the structure of a ULC-module from the tautological module ULC — we obtain the *derived module* (cf. [Sect. 2.2.3.1](#)) of the module ULC. Also, the map

$$\text{ULC} \times \text{ULC} : V \mapsto \text{ULC}(V) \times \text{ULC}(V)$$

inherits a ULC-module structure, cf. [Def. 2.53](#).

The constructors of our example languages are, accordingly, *morphisms of modules*:

2.47 Example ([Ex. 2.46](#) cont.): The map

$$V \mapsto \text{App}_V : \text{ULC}(V) \times \text{ULC}(V) \rightarrow \text{ULC}(V)$$

satisfies the diagram of [Rem. 2.44](#) and is hence a morphism of ULC-modules from $\text{ULC} \times \text{ULC}$ to ULC. The map

$$V \mapsto \text{Abs}_V : \text{ULC}(V') \rightarrow \text{ULC}(V)$$

is a morphism of ULC-modules from ULC' to ULC. Later we consider this example using an alternative definition of module morphism (cf. [Def. 2.73](#)) and explain in detail the meaning of its commutative diagrams for the constructors App and Abs, cf. [Ex. 2.74](#).

2.2.2. Constructions on Monads and Modules

We present some constructions of modules which will be used in the next section. They were previously defined in Zsidó's thesis [[Zsi10](#)] and works of Hirschowitz and Maggesi [[HM07a](#), [HM10a](#)].

2.48 Definition (Tautological Module): Given a monad (\mathcal{C}, T) , we call (T, μ_T) (or simply T) the *tautological module* $(T, \mu_T) : (\mathcal{C}, T) \rightarrow (\mathcal{C}, \text{Id})$.

2.49 Definition (Constant and Terminal Module): Given a monad (\mathcal{C}, T) and a category \mathcal{D} with an object $d \in \mathcal{D}$, the constant functor $F_d : \mathcal{C} \rightarrow \mathcal{D}$ mapping any object of \mathcal{C} to $d \in \mathcal{D}$ and any morphism to the identity on d yields a module

$$(F_d, \text{id}) : (\mathcal{C}, T) \rightarrow (\mathcal{D}, \text{Id}) .$$

In particular, if \mathcal{D} has a terminal object $1_{\mathcal{D}}$, then the constant module $(F_{1_{\mathcal{D}}}, \text{id})$ is terminal in $\text{Mod}(T, \mathcal{D})$.

2.50 Remark: Given a monad (\mathcal{C}, T) , a T –module (M, σ) with codomain category \mathcal{D} and a functor $F : \mathcal{D} \rightarrow \mathcal{E}$, then the pair $(F \circ M, F\sigma)$ is a T –module with codomain category \mathcal{E} . For $(M, \sigma) := (T, \mu_T)$ and $F := F_e$ for some $e \in \mathcal{E}$ one obtains the constant module as above.

2.51 Definition (Pullback Module): Let (\mathcal{C}, T) and (\mathcal{D}, T') be monads over \mathcal{C} and \mathcal{D} , respectively. Given a morphism of monads $(F, \gamma) : (\mathcal{C}, T) \rightarrow (\mathcal{D}, T')$ and a T' –module (M, σ) with codomain \mathcal{E} , we call *pullback of M along (F, γ)* the T –module $(F, \gamma)^*(M, \sigma) := (M, \sigma) \circ (F, \gamma)$.

2.52 Definition (Module Morphism induced by a Monad Morphism): With the same notation as in the previous example, the monad morphism (F, γ) induces a morphism of T –modules — which we call γ as well —

$$\gamma : (F, \text{id}) \circ T \Rightarrow (F, \gamma)^*(T', \mu_{T'})$$

as in

$$\begin{array}{ccc} & (\mathcal{C}, T) & \\ (T, \mu_T) \swarrow & & \searrow (F, \gamma) \\ (\mathcal{C}, \text{Id}) & \xRightarrow{\gamma} & (\mathcal{D}, T') \\ (F, \text{id}) \searrow & & \swarrow (T', \mu_{T'}) \\ & (\mathcal{D}, \text{Id}) & \end{array} .$$

Note that the above diagram can be read as a structure–enriched version of the square diagram specifying the type of γ in [Def. 2.38](#).

2.53 Definition (Product Module): Suppose the category \mathcal{D} is equipped with a product. Given any monad (\mathcal{C}, T) , the product of \mathcal{D} lifts to a product on the category $\text{Mod}(T, \mathcal{D})$ of T –modules with codomain \mathcal{D} .

2.2.3. Monads on Set Families

We are particularly interested in monads over families of sets and monad morphisms over retyping functors.

2.2.3.1. Derivation

Roughly speaking, a binding constructor makes free variables disappear. Its inputs are hence terms “with (one or more) additional free variables” compared to the output, i.e. terms in an *extended context*. Context extension is captured mathematically by *derivation*:

let T be a set and $u \in T$ an element of T . We define $D(u)$ to be the object of Set^T such that

$$D(u)(u) = \{*\} \quad \text{and} \quad D(u)(t) = \emptyset \text{ for } t \neq u .$$

We enrich the object V of Set^T with respect to u by setting

$$V^{*u} := V + D(u) ,$$

that is, we add a fresh variable of type u . This yields a monad $(_)^{*u}$ on Set^T .

2.54 Definition (Derivation Monad Morphism): Given any monad P on Set^T , we define a monad endomorphism on P over the functor $V \mapsto V^{*u}$. On a set family $V \in \text{Set}^T$ its natural transformation γ is defined as the coproduct map

$$\gamma_V := [P(\text{inl}), \eta \circ \text{inr}] : (PV)^{*u} \rightarrow P(V^{*u}) , \quad (2.2.1)$$

where $[\text{inl}, \text{inr}] = \text{id} : V^{*u} \rightarrow V^{*u}$.

2.55 Definition: Given a monad P over Set^T and a P -module M , we call M^u the module obtained as the composition $M \circ (_)^{*u}$.

2.56 Example: We consider TLC (cf. [Ex. 2.37](#)) as the tautological module over itself. Given any element $s \in T_{\text{TLC}}$, the derived module with respect to s ,

$$\text{TLC}^s : V \mapsto \text{TLC}(V^{*s}) ,$$

assigns to any type family V — the *context* — the type family of terms of TLC over V enriched with one additional variable of sort s .

More generally, given a natural transformation as in [Rem. 2.30](#),

$$\tau : 1 \Rightarrow \mathcal{T}U_n : \mathcal{T}\text{Set}_n \rightarrow \text{Set} ,$$

we can derive, with respect to τ , any module defined on a category of the form Set_n^T for any set T :

2.57 Definition (Derived Module): Let $\tau : 1 \rightarrow \mathcal{T}U_n$ be a natural transformation. Given a set T and a monad P on Set_n^T , the functor $(_)^{*\tau} : (T, V, \mathbf{t}) \mapsto (T, V^{*\tau(T, V, \mathbf{t})}, \mathbf{t})$ is given the structure of a morphism of monads as in [Disp. \(2.2.1\)](#). Given any P -module M , we call *derivation of M with respect to τ* the module $M^\tau := M \circ (_)^{*\tau}$.

2.58 Remark: In the preceding definition the natural transformation $\tau : 1 \rightarrow \mathcal{T}U_n$ supplies more data than necessary, since we only evaluate it on families of sets indexed by the fixed set T . However, in [Sect. 3.3](#) we derive different modules — each defined on a category Set_n^T with varying sets T — with respect to one and the same natural transformation τ .

2.2.3.2. Fibres

Given a typed language over a nonempty set of types T , we occasionally want to pick terms of a specific type $u \in T$. Let \mathcal{D} be a category — think of \mathcal{D} as the category Set — and $V \in \mathcal{D}^T$ a T -indexed family, e.g., of terms of said language. Then picking “terms of type $u \in T$ ” corresponds to projecting to the fibre $V(u)$.

Given a monad P on a category \mathcal{C} and a P -module M towards \mathcal{D}^T , we define the *fibre module of M with respect to $u \in T$* to be the module which associates the fibre $M(c)(u)$ to any object $c \in \mathcal{C}$. This construction is expressed via postcomposition with a particular module: we define the *fibre with respect to $u \in T$* to be the monad morphism

$$(_)(u), \text{id} : (\mathcal{D}^T, \text{Id}) \rightarrow (\mathcal{D}, \text{Id})$$

over the functor $V \mapsto V(u)$. Postcomposition of the module M with this module then precisely yields the fibre module $[M]_u$ of M with respect to $u \in T$. Analogously to derivation we define the fibre with respect to a natural transformation:

2.59 Definition (Fibre Module): Let the natural transformation τ be as in [Def. 2.57](#). We call *fibre with respect to τ* the monad morphism

$$(_)_{\tau} : V \mapsto V(\tau_V) : (\mathcal{D}_n^T, \text{Id}) \rightarrow (\mathcal{D}, \text{Id})$$

over the functor $V \mapsto V_{\tau(V)}$. Given a module M towards \mathcal{D}_n^T (over some monad P), we call the *fibre module of M with respect to τ* the module $[M]_{\tau} := (_)_{\tau} \circ M$.

2.60 Example: We consider TLC as the tautological module over itself. Given any element $t \in \mathcal{T}$, the fibre module with respect to t , denotes the set of terms of TLC of type t in context V :

$$[\text{TLC}]_t : V \mapsto \text{TLC}(V)_t .$$

2.61 Example: Consider the monad $\text{TLC} : \text{Set}^{\text{TLC}} \rightarrow \text{Set}^{\text{TLC}}$ of [Ex. 2.37](#). The two operations of derivation (cf. [Ex. 2.56](#)) and fibre (cf. [Ex. 2.60](#)) can be combined, yielding a module over TLC with carrier

$$V \mapsto \text{TLC}_t^s(V) := \text{TLC}(V^{*s})_t .$$

This module is actually the domain module of the abstraction constructor, cf. [Ex. 2.62](#). The product of modules yields our final example: for any $s, t \in T_{\text{TLC}}$, the domain of the application $\text{App}(s, t)$ of simply-typed lambda calculus is a module over TLC,

$$[\text{TLC}]_{s \rightsquigarrow t} \times [\text{TLC}]_s : V \mapsto \text{TLC}(V)_{s \rightsquigarrow t} \times \text{TLC}(V)_s .$$

2.62 Example ([Ex. 2.61](#) cont.): Given $s, t \in T_{\text{TLC}}$, the map

$$\text{App}(s, t) : V \mapsto \text{App}_V(s, t) : \text{TLC}(V)_{s \rightsquigarrow t} \times \text{TLC}(V)_s \rightarrow \text{TLC}(V)_t$$

satisfies the diagram of the preceding definition and is hence a morphism of modules. In the same way the constructor $\text{Abs}(s, t)$ is a morphism of modules; we have

$$\begin{aligned} \text{App}(s, t) &: [\text{TLC}]_{s \rightsquigarrow t} \times [\text{TLC}]_s \rightarrow [\text{TLC}]_t \\ \text{Abs}(s, t) &: [\text{TLC}^s]_t \rightarrow [\text{TLC}]_{s \rightsquigarrow t} . \end{aligned}$$

The pullback operation commutes with products, derivations and fibres:

2.63 Remark: Let (\mathcal{C}, P) and (\mathcal{D}, Q) be monads, and let $\rho : P \rightarrow Q$ be a monad morphism. Let M be a Q -module with codomain \mathcal{E} . Suppose T is a set, and let $u \in T$ be an element of T .

1. More specifically, let Q be a monad on Set^T . Then

$$\rho^*(M^u) = (\rho^*M)^u .$$

2. More specifically, let $\mathcal{E} = \mathcal{C}^T$. Then

$$\rho^*[M]_u = [\rho^*M]_u .$$

3. Let N be another Q -module with codomain \mathcal{E} and suppose \mathcal{E} is equipped with a product. Then the pullback functor is cartesian:

$$\rho^*(M \times N) = \rho^*M \times \rho^*N .$$

The first two properties are just instances of associativity of composition of monad morphisms.

2.64 Remark: In Coq the equality of modules is not as trivial as in informal mathematics, since there are two different notions of equality: *definitional equality*, also called *convertibility*, and *propositional equality*. While the latter is to be proved by the user, the former is computed by the system and thus cannot be influenced by the user.

While the above equalities of [Rem. 2.63](#) hold propositionally (using appropriate axioms, such as proof irrelevance), they do not hold definitionally. The consequences of this lack of definitional equality are discussed in [Sect. 7.1.2](#). In summary, in our formalization, monads, modules and module morphisms behave more like in a *bicategory* rather than in a strict 2-category.

2.3. Alternative Definitions for Monads & Modules

Monads can be defined in terms of the Kleisli operation (cf. [Rem. 2.35](#)) instead of the natural transformation μ of [Def. 2.33](#). A similar alternative definition exists for modules.

2. Category–Theoretic Constructions

In this section we state those alternative definitions in full detail, for several reasons: firstly, the alternative definition of monad is well-known for its prominent use in the `HASKELL` programming language. Secondly, it is also the definition we chose to implement in the proof assistant `Coq`. Furthermore, it is also this alternative definition which generalizes to *relative monads* (cf. [Def. 2.75](#)), that is, monads that are not necessarily endofunctors.

2.65 Definition (Alt. Def. for Monad ([Def. 2.33](#)), [Code 6.10](#)): A monad T over a category \mathcal{C} (in Kleisli form) is given by

- a map $T : \mathcal{C} \rightarrow \mathcal{C}$ on the objects of \mathcal{C} , carrying the same name as the monad,
- for each object c of \mathcal{C} , a morphism $\eta_c \in \mathcal{C}(c, Tc)$ and
- for all objects c and d of \mathcal{C} , a Kleisli map

$$\sigma_{c,d} : \mathcal{C}(c, Td) \rightarrow \mathcal{C}(Tc, Td)$$

such that the following diagrams commute for all suitable morphisms f and g :

$$\begin{array}{ccc} \begin{array}{ccc} c & \xrightarrow{\eta_c} & Tc \\ & \searrow f & \downarrow \sigma(f) \\ & & Td \end{array} & \begin{array}{ccc} Tc & & \\ & \searrow \sigma(\eta_c) & \\ & \text{id} & \searrow \\ & & Tc \end{array} & \begin{array}{ccc} Tc & \xrightarrow{\sigma(f)} & Td \\ & \searrow \sigma(\sigma(g) \circ f) & \downarrow \sigma(g) \\ & & Te \end{array} \end{array}$$

We also refer to the Kleisli map as “substitution map”: when \mathcal{C} is instantiated, for example, by the category of sets and TX is a set of terms with free variables in the set X , then *simultaneous substitution* as Kleisli map turns T into a monad. In this case the diagrams express the well-known substitution properties [\[AR99\]](#). More precisely, the first diagram determines the value of substitution on variables, the second diagram states that substituting each variable by itself in a term does not change the term, and the third diagram shows how two consecutive substitutions can be expressed by just one substitution. Inspired by `HASKELL` syntax, we frequently use the infix symbol $\gg=$ to denote simultaneous substitution (or more generally, Kleisli maps): given a term $M \in TX$ with free variables in X and $f : X \rightarrow TY$, then

$$M \gg= f := \sigma(f)(M)$$

denotes the term obtained by replacing any free variable $x \in X$ occurring in M by its image $f(x) \in TY$, yielding a term in TY .

The following remarks recover the definition of monad given in [Def. 2.33](#) from the definition of [Def. 2.65](#).

2.66 Remark *Functoriality for Monads in Kleisli Form*, [Code 6.11](#): Given a monad T over \mathcal{C} as in [Def. 2.65](#) and a morphism $f : c \rightarrow d$ in \mathcal{C} , we equip T with a functorial structure by setting

$$T(f) := \text{lift}_T(f) := \sigma(\eta_d \circ f) \quad .$$

2.67 Remark *Naturality of η and Multiplication for Monads in Kleisli form*: Given a monad in Kleisli form T , the family of morphisms $\eta = (\eta_c : \mathcal{C}(c, Tc))_{c \in \mathcal{C}}$ is natural with respect to the functorial structure defined in [Rem. 2.66](#). A multiplication $\mu : T^2 \rightarrow T$ can be defined as substitution with identity:

$$\mu_c := \sigma(\text{id}_{Tc}) : TTc \rightarrow Tc \quad .$$

Naturality of μ is a consequence of the axioms for monads in Kleisli form. Finally, the monad multiplication μ thus defined is compatible with the unit η in the sense of [Def. 2.33](#).

2.68 Remark *Naturality of Substitution*: Given a monad in Kleisli form T over \mathcal{C} , then its substitution σ is natural in c and d . For naturality in c we check that the diagram

$$\begin{array}{ccccc} c & \mathcal{C}(c, Td) & \xrightarrow{\sigma_{c,d}} & \mathcal{C}(Tc, Td) \\ f \downarrow & \uparrow f^* & & \uparrow (Tf)^* \\ c' & \mathcal{C}(c', Td) & \xrightarrow{\sigma_{c',d}} & \mathcal{C}(Tc', Td) \end{array}$$

commutes, where $f^*(h) := h \circ f$. Given $g \in \mathcal{C}(c', Td)$, we have

$$\begin{aligned} \sigma(g) \circ Tf &= \sigma(g) \circ \sigma(\eta_{c'} \circ f) \\ &\stackrel{3}{=} \sigma(\sigma(g) \circ \eta_{c'} \circ f) \\ &\stackrel{1}{=} \sigma(g \circ f) \quad , \end{aligned}$$

where the numbers correspond to the diagrams of [Def. 2.65](#) used to rewrite in the respective step. Similarly we check naturality in d . Writing $h_*(g) := h \circ g$, the diagram

$$\begin{array}{ccccc} d & \mathcal{C}(c, Td) & \xrightarrow{\sigma_{c,d}} & \mathcal{C}(Tc, Td) \\ h \downarrow & (Th)_* \downarrow & & \downarrow (Th)_* \\ d' & \mathcal{C}(c', Td) & \xrightarrow{\sigma_{c',d}} & \mathcal{C}(Tc', Td) \end{array}$$

commutes: given $g \in \mathcal{C}(c, Td)$, we have

$$\begin{aligned} Th \circ \sigma(g) &= \sigma(\eta_{d'} \circ h) \circ \sigma(g) \\ &\stackrel{3}{=} \sigma(\sigma(\eta_{d'} \circ h) \circ g) \\ &= \sigma(Th \circ g) \quad . \end{aligned}$$

2.69 Definition (Morphism of Monads, Alt. to [Def. 2.38](#), [Code 6.12](#)): Let (\mathcal{C}, T) and (\mathcal{D}, T') be two monads. A *colax morphism of monads* $\tau : T \rightarrow T'$ is given by

- a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ and
- for any $c \in \mathcal{C}$, a morphism $\tau_c : FTc \rightarrow T'Fc$

such that the following diagrams commute for all suitable morphisms f :

$$\begin{array}{ccc}
 FTc & \xrightarrow{F(\sigma^T(f))} & FTd \\
 \tau_c \downarrow & & \downarrow \tau_d \\
 T'Fc & \xrightarrow{\sigma^{T'}(\tau_d \circ Ff)} & T'Fd
 \end{array} , \qquad
 \begin{array}{ccc}
 Fc & \xrightarrow{F\eta_c^T} & FTc \\
 \eta_{Fc}^{T'} \searrow & & \downarrow \tau_c \\
 & & T'Fc
 \end{array} .$$

2.70 Remark: Naturality of the family $(\tau_c)_{c \in \mathcal{C}}$ of a colax morphism of monads as in the preceding definition is provable from the other axioms, yielding a natural transformation

$$\tau : FT \rightarrow T'F .$$

Here we use [Rem. 2.66](#) by considering T and T' as functors. The naturality of τ is proved in Lemma `colax_Monad_Hom_NatTrans` in the Coq library.

2.71 Definition (Module, Alt. to [Rem. 2.44](#), [Code 6.14](#)): Let \mathcal{D} be a category. A *module* M over T with codomain \mathcal{D} is given by

- a map $M : \mathcal{C} \rightarrow \mathcal{D}$ on the objects of the categories involved and
- for all objects c, d of \mathcal{C} , a map

$$\varsigma_{c,d} : \mathcal{C}(c, Td) \rightarrow \mathcal{D}(Mc, Md)$$

such that the following diagrams commute for all suitable morphisms f and g :

$$\begin{array}{ccc}
 Mc & \xrightarrow{\varsigma(f)} & Md \\
 \varsigma(\sigma(g) \circ f) \searrow & & \downarrow \varsigma(g) \\
 & & Me,
 \end{array}
 \qquad
 \begin{array}{ccc}
 Mc & & \\
 \searrow \varsigma(\eta_c) & \searrow \text{id} & \\
 & & Mc.
 \end{array}$$

2.72 Remark: Functoriality for such a module M is defined similarly to that for monads: for any morphism $f : c \rightarrow d$ in \mathcal{C} we set

$$M(f) := \text{mlift}_M(f) := \varsigma(\eta^T \circ f) .$$

A *module morphism* is a family of morphisms that is compatible with module substitution:

2.73 Definition (Module Morphism, Alt. to [Rem. 2.44](#), [Code 6.15](#)): Let M and N be two modules over T with codomain \mathcal{D} . A *morphism of T -modules* from M to N is given by a family of morphisms $\rho_c \in \mathcal{D}(Mc, Nc)$ such that for all morphisms $f \in \mathcal{C}(c, Td)$ the following diagram commutes:

$$\begin{array}{ccc} Mc & \xrightarrow{\zeta^M(f)} & Md \\ \rho_c \downarrow & & \downarrow \rho_d \\ Nc & \xrightarrow{\zeta^N(f)} & Nd. \end{array}$$

A module morphism $M \rightarrow N$ also constitutes a natural transformation between the functors M and N induced by the modules, cf. `Module_Hom_NatTrans`.

2.74 Example ([Ex. 2.47](#) cont.): We consider [Ex. 2.47](#) under the alternative definition of module morphism. The map

$$V \mapsto \text{App}_V : \text{ULC}(V) \times \text{ULC}(V) \rightarrow \text{ULC}(V)$$

satisfies the diagram of the preceding definition and is hence a morphism of ULC-modules from $\text{ULC} \times \text{ULC}$ to ULC . The property of being a module morphism expresses distributivity of substitution for any substitution map $f : X \rightarrow \text{ULC}(Y)$:

$$\text{App}(M, N) \gg f = \text{App}(M \gg f, N \gg f) .$$

Similarly, the map

$$V \mapsto \text{Abs}_V : \text{ULC}(V') \rightarrow \text{ULC}(V)$$

is a morphism of ULC-modules from ULC' to ULC . For $f : X \rightarrow \text{ULC}(Y)$ as before, the commutative diagram here expresses the equation

$$\text{Abs}(M) \gg f = \text{Abs}(M \gg f') ,$$

where $f' : X' \rightarrow \text{ULC}(Y')$ is obtained by shifting the map f to account for the extended context under the binder `Abs`.

Modules on P with codomain \mathcal{D} and morphisms between them form a category called $\text{Mod}(P, \mathcal{D})$ (in the library: `MOD P D`), similar to the category of monads.

2.4. Relative Monads and Modules

The functors underlying the monads presented in the preceding section all are endofunctors. This is enforced by the type of monadic multiplication and substitution. *Relative monads* were defined by Altenkirch et al. [ACU10] to overcome this restriction. One of their motivations was to consider the untyped lambda calculus over *finite* contexts as a monad-like structure — similar to the monad structure on the lambda calculus over arbitrary contexts exhibited by Altenkirch and Reus [AR99].

We review the definition of relative monads and define suitable *colax morphisms of relative monads*. Afterwards we define *modules over relative monads* and port the constructions on modules over monads (cf. Sects. 2.2.2 and 2.2.3) to modules over relative monads.

2.4.1. Definitions

We review the definition of relative monad as given by Altenkirch et al. [ACU10] and define suitable morphisms for them. As an example we consider the lambda calculus as a relative monad from sets to preorders, on the functor Δ (cf. Def. 2.13). Afterwards we define *modules over relative monads* and carry over the constructions on modules over regular monads of the preceding section to modules over relative monads.

The definition of relative monads is analogous to that of monads in Kleisli form (cf. Def. 2.65), except that the underlying map of objects is between *different* categories. Thus, for the operations to remain well-typed, one needs an additional “mediating” functor, in the following usually called F , which is inserted wherever necessary:

2.75 Definition (Relative Monad, [ACU10], Code 6.16): Given categories \mathcal{C} and \mathcal{D} and a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, a *relative monad* $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ on F is given by the following data:

- a map $P : \mathcal{C} \rightarrow \mathcal{D}$ on the objects of \mathcal{C} ,
- for each object c of \mathcal{C} , a morphism $\eta_c \in \mathcal{D}(Fc, Pc)$ and
- for each two objects c, d of \mathcal{C} , a *substitution map*

$$\sigma_{c,d} : \mathcal{D}(Fc, Pd) \rightarrow \mathcal{D}(Pc, Pd)$$

such that the following diagrams commute for all suitable morphisms f and g :

$$\begin{array}{ccc}
 \begin{array}{ccc} Fc & \xrightarrow{\eta_c} & Pc \\ & \searrow f & \downarrow \sigma(f) \\ & & Pd \end{array} &
 \begin{array}{ccc} Pc & & \\ & \searrow \text{id} & \downarrow \sigma(\eta_c) \\ & & Pc \end{array} &
 \begin{array}{ccc} Pc & \xrightarrow{\sigma(f)} & Pd \\ & \searrow \sigma(\sigma(g) \circ f) & \downarrow \sigma(g) \\ & & Pe \end{array}
 \end{array}$$

2.76 Example (Lambda Calculus over Finite Contexts, [ACU10]): Altenkirch et al. [ACU10] consider the untyped lambda calculus as a relative monad on the functor $J : \text{Fin}_{\text{skel}} \rightarrow \text{Set}$. Here the category Fin_{skel} is the category of finite cardinals, i.e. the skeleton of the category Fin of *finite* sets and maps between finite sets.

2.77 Remark: Relative monads on the identity functor $\text{Id} : \mathcal{C} \rightarrow \mathcal{C}$ precisely correspond to monads as presented in Def. 2.65.

2.78 Notation: For this section we reserve the term “monad” for monads as defined in Def. 2.65, and explicitly state the “relative” when talking about relative monads. In later sections we sometimes omit the attribute “relative” and instead refer to traditional monads (i.e. with $F = \text{Id}$) as *regular* or *plain* monads.

2.79 Remark *Restricting a Monad yields a Relative Monad*, [ACU10]: Given a monad T on \mathcal{D} and a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, then the monad T restricts to a relative monad $T^b : \mathcal{C} \xrightarrow{F} \mathcal{D}$ by precomposing with F .

2.80 Remark *Relative Monads are functorial*, Code 6.17: Given a monad P over $F : \mathcal{C} \rightarrow \mathcal{D}$ and a morphism $f : c \rightarrow d$ in \mathcal{C} , a functorial structure (rlift) for P is defined by setting

$$P(f) := \text{lift}_P(f) := \sigma (\eta \circ Ff) \quad .$$

The functor axioms are easily proved from the monadic axioms.

2.81 Remark *Relative Monads as Monoids in a Functor Category*, [ACU10]: A monad (T, η, μ) over a category \mathcal{C} is the same as a monoid object in the functor category $[\mathcal{C}, \mathcal{C}]$, where the monoidal structure is given by functor composition. Altenkirch et al. [ACU10] recover a similar characterization for relative monads on a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, provided that the left Kan extension along F ,

$$\text{Lan}_F : [\mathcal{C}, \mathcal{D}] \rightarrow [\mathcal{D}, \mathcal{D}] \quad ,$$

exists: they define a lax monoidal structure on $[\mathcal{C}, \mathcal{D}]$ by

$$\begin{aligned} (\cdot^F) : [\mathcal{C}, \mathcal{D}] \times [\mathcal{C}, \mathcal{D}] &\rightarrow [\mathcal{C}, \mathcal{D}] \\ (H, G) &\mapsto H \cdot^F G := \text{Lan}_F H \circ G \quad . \end{aligned}$$

They then show that relative monads on F correspond precisely to lax monoid objects in $([\mathcal{C}, \mathcal{D}], \cdot^F)$. Besides, they show that under some coherence conditions, this result can be sharpened to obtain a strict monoidal structure, where relative monads correspond to proper monoids with respect to this structure. Under the same assumptions, a relative monad P on $F : \mathcal{C} \rightarrow \mathcal{D}$ can be extended to a traditional monad P^\sharp on \mathcal{D} , yielding an adjunction $(_)^\sharp \dashv (_)^b$. This adjunction furthermore is a coreflection.

2. Category-Theoretic Constructions

2.82 Remark *Naturality of Substitution*: Analogously to [Rem. 2.68](#), the substitution $\sigma = (\sigma_{c,d})$ of a relative monad P on a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ is binatural.

We are interested in monads on the category \mathbf{Set} of sets and relative monads on $\Delta : \mathbf{Set} \rightarrow \mathbf{Pre}$ as well as their relationship:

2.83 Lemma (Relative Monads on Δ and Monads on \mathbf{Set}): *Let P be a relative monad on $\Delta : \mathbf{Set} \rightarrow \mathbf{Pre}$ (cf. [Def. 2.13](#)). By postcomposing with the forgetful functor $U : \mathbf{Pre} \rightarrow \mathbf{Set}$ we obtain a monad*

$$UP : \mathbf{Set} \rightarrow \mathbf{Set} .$$

The substitution is defined, for $m : X \rightarrow UPY$ by setting

$$U\sigma : m \mapsto U\left(\sigma\left(\varphi^{-1}m\right)\right) ,$$

as indicated by the diagram

$$\begin{array}{ccc} \mathbf{Set}(X, UPY) & \xrightarrow{U\sigma} & \mathbf{Set}(UPX, UPY) \\ \varphi^{-1} \downarrow & & \uparrow U \\ \mathbf{Pre}(\Delta X, PY) & \xrightarrow{\sigma} & \mathbf{Pre}(PX, PY) \end{array}$$

making use of the adjunction φ of [Lem. 2.18](#).

Conversely, to any monad T over \mathbf{Set} , given as a Kleisli triple, we associate a relative monad over Δ by postcomposing with Δ . The substitution map $\Delta\sigma$ is defined, for $m : \Delta X \rightarrow \Delta TY$, as the following composition:

$$\begin{array}{ccc} \mathbf{Pre}(\Delta X, \Delta TY) & \xrightarrow{\Delta\sigma} & \mathbf{Pre}(\Delta TX, \Delta TY) \\ U \downarrow & & \uparrow \varphi^{-1} \\ \mathbf{Set}(X, TY) & \xrightarrow{\sigma} & \mathbf{Set}(TX, TY) \end{array}$$

The maps thus defined are object functions of an adjunction between monads on sets and relative monads on Δ , cf. [Lem. 4.5](#).

The above construction actually is an instance of a more general construction:

2.84 Lemma (Monads from Relative Monads and conversely): *Let $F : \mathcal{C} \rightleftarrows \mathcal{D} : G$ be an adjunction with a family of isomorphisms*

$$\varphi_{X,Y} : \mathcal{D}(FX, Y) \cong \mathcal{C}(X, GY) : \varphi_{X,Y}^{-1} .$$

1. Given a relative monad $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ with unit η and substitution σ , we define a monad P^+ on \mathcal{C} by setting

$$\begin{aligned} P^+(c) &:= GPc, \\ \eta_c^+ &:= \varphi(\eta_c) : \mathcal{C}(c, GPc) \text{ and} \\ \sigma_{c,d}^+(f) &:= G\left(\sigma(\varphi^{-1}(f))\right). \end{aligned}$$

2. Let furthermore $GF = \text{Id}$ be the identity on \mathcal{C} . Given a monad (P, η, σ) on \mathcal{C} , we define a relative monad $P^- : \mathcal{C} \xrightarrow{F} \mathcal{D}$ by setting

$$\begin{aligned} P^-(c) &:= FPc, \\ \eta_c^- &:= F(\eta_c) \text{ and} \\ \sigma_{c,d}^- &:= \varphi^{-1}(\sigma(Gf)). \end{aligned}$$

Proof. We check the commutativity of the corresponding diagrams:

1. for the data (P^+, η^+, σ^+) :

- $\sigma^+(f) \circ \eta_c = G(\sigma(\varphi^{-1}f)) \circ \varphi(\eta_c) = \varphi(\sigma(\varphi^{-1}f) \circ \eta_c) = \varphi\varphi^{-1}f = f$
- $\sigma^+(\eta_c^+) = G(\sigma(\varphi^{-1}(\varphi(\eta_c)))) = G(\sigma(\eta_c)) = G\text{id} = \text{id}$
-

$$\begin{aligned} \sigma^+(g) \circ \sigma^+(f) &= G\sigma\varphi^{-1}g \circ G\sigma\varphi^{-1}f \\ &= G(\sigma(\varphi^{-1}g) \circ \sigma(\varphi^{-1}f)) \\ &= G(\sigma(\sigma(\varphi^{-1}g) \circ \varphi^{-1}f)) \\ &= G(\sigma(\varphi^{-1}(G(\sigma(\varphi^{-1}g)) \circ f))) \\ &= \sigma^+(\sigma^+(g) \circ f) \end{aligned}$$

2. for the data (P^-, η^-, σ^-) :

- $\sigma^-(f) \circ \eta_c^- = \varphi^{-1}(\sigma(Gf)) \circ F\eta_c = \varphi^{-1}(\sigma(Gf) \circ \eta_c) = \varphi^{-1}(Gf) = f$
- $\sigma^-(\eta_c^-) = \varphi^{-1}(\sigma(G\eta_c^-)) = \epsilon_{FPC} \circ F(\sigma(GF\eta_c)) = \epsilon_{FPC} \circ F(\sigma(\eta_c)) = \epsilon_{FPC} \circ F\text{id} = \text{id}$

•

$$\begin{aligned}
 \sigma^-(\sigma^-g \circ f) &= \varphi^{-1}\left(\sigma\left(G(\sigma^-g \circ f)\right)\right) \\
 &= \varphi^{-1}\left(\sigma\left(G\sigma^-g \circ Gf\right)\right) \\
 &= \varphi^{-1}(\sigma(\sigma Gg \circ Gf)) \\
 &= \varphi^{-1}(\sigma Gg \circ \sigma Gf) \\
 &= \epsilon_F \circ F(\sigma Gg \circ \sigma Gf) \\
 &= \epsilon_F \circ F\sigma Gg \circ F\sigma Gf \\
 &= \epsilon_F \circ F\sigma Gg \circ \epsilon_F \circ F\sigma Gf \\
 &= \varphi^{-1}\sigma Gg \circ \varphi^{-1}\sigma Gf \\
 &= \sigma^-g \circ \sigma^-f
 \end{aligned}$$

□

This construction is functorial, and yields an adjunction between a category of monads on \mathcal{C} and relative monads on F . Details will be reported elsewhere.

2.85 Example (Lambda Calculus as Relative Monad on Δ): Consider the set of all lambda terms indexed by their set of free variables as defined in [Ex. 1.2](#). We write λM and MN for $\text{Abs } M$ and $\text{App } MN$, respectively. We equip each $\text{ULC}(V)$ with a preorder taken as the reflexive–transitive closure of the relation generated by the rule

$$(\lambda M)N \leq M[* := N]$$

and its propagation into subterms. This defines a monad ULCBETA from sets to pre-orders over the functor Δ ,

$$\text{ULC}_\beta : \text{Set} \xrightarrow{\Delta} \text{Pre}.$$

The family η^{ULC} is given by the constructor Var , and the substitution map

$$\sigma_{X,Y} : \text{Pre}(\Delta(X), \text{ULC}_\beta(Y)) \rightarrow \text{Pre}(\text{ULC}_\beta(X), \text{ULC}_\beta(Y))$$

is given by capture–avoiding simultaneous substitution. Via the adjunction of [Lem. 2.18](#) the substitution can also be read as

$$\sigma_{X,Y} : \text{Set}(X, \text{ULC}(Y)) \rightarrow \text{Pre}(\text{ULC}_\beta(X), \text{ULC}_\beta(Y)) .$$

2.86 Remark about Substitution: The substitution in [Ex. 2.85](#) is compatible with the order on terms in the following sense:

1. $M \leq N$ implies $M[* := A] \leq N[* := A]$ and

2. $A \leq B$ implies $M[* := A] \leq M[* := B]$.

The first implication is a general fact for any relative monad P on Δ : it is a special case of $\sigma_{X,Y}(f)$ being a morphism in the category Pre for any $f \in \text{Pre}(\Delta V, PW)$. The second monotony property, however, is *false* in general. As an example, consider the monad given by

$$\begin{aligned} F(V) ::= & \quad \text{Var} : V \rightarrow F(V) \\ & | \quad \perp : F(V) \\ & | \quad (\Rightarrow) : F(V) \times F(V) \rightarrow F(V) \end{aligned}$$

equipped with a preorder which is contravariant in the first argument of the arrow constructor \Rightarrow . Substituting in this position, the first argument of (\Rightarrow) , does in fact *reverse* the order on terms, i.e. we obtain (using \Rightarrow infix)

$$A < B \quad \text{implies} \quad (* \Rightarrow M)[* := B] < (* \Rightarrow M)[* := A] .$$

A different definition of monad which would enforce the second implication to hold — and hence not include the example F — can be given easily by considering Pre as a 2-category enriched over itself: given morphisms $f, g \in \text{Pre}(X, Y)$ we say that there is precisely one 2-cell

$$f \Rightarrow g \quad \text{iff} \quad f \leq g \quad \text{iff} \quad \forall x : X, f(x) \leq g(x) .$$

A monad P would then have to be equipped with a substitution action that is given, for any two sets V and W , by a *functor* (of preorders)

$$\sigma_{V,W} : \text{Pre}(\Delta V, PW) \rightarrow \text{Pre}(PV, PW) .$$

Definition 2.110 explains one of the consequences of our monadic substitution lacking “higher-order monotonicity”.

We generalize the definition of colax monad morphisms to relative monads:

2.87 Definition (Colax Morphism of Relative Monads, [Code 6.18](#)): Let $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ and $Q : \mathcal{C}' \xrightarrow{F'} \mathcal{D}'$ be two relative monads. A *colax morphism of relative monads* from P to Q is given by a quadruple (G, G', N, τ) consisting of a functor $G : \mathcal{C} \rightarrow \mathcal{C}'$ and a functor $G' : \mathcal{D} \rightarrow \mathcal{D}'$ as well as a natural transformation $N : F'G \rightarrow G'F$ as in

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{F} & \mathcal{D} \\ \downarrow G & \nearrow N & \downarrow G' \\ \mathcal{C}' & \xrightarrow{F'} & \mathcal{D}' \end{array}$$

2. Category–Theoretic Constructions

and a natural transformation $\tau : G' \circ P \rightarrow Q \circ G$ as in

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{P} & \mathcal{D} \\ G \downarrow & \not\parallel_{\tau} & \downarrow G' \\ \mathcal{C}' & \xrightarrow{Q} & \mathcal{D}', \end{array}$$

such that the following diagrams commute for all suitable morphisms f :

$$\begin{array}{ccc} G'Pc \xrightarrow{G'\sigma^P(f)} G'Pd & & F'Gc \xrightarrow{Nc} G'Fc \xrightarrow{G'\eta_c^P} G'Pc \\ \tau_c \downarrow & & \searrow \eta_{Gc}^Q \quad \downarrow \tau_c \\ QGc \xrightarrow{\sigma^Q(\tau_d \circ G'f \circ Nc)} QGd & & QGc. \end{array}$$

2.88 Remark: Naturality of τ in the preceding definition is actually a consequence of the commutative diagrams of [Def. 2.87](#), cf. `Lemma colax_RM Monad_Hom_NatTrans` in the Coq library.

2.89 Remark: In [Chapt. 5](#) we are going to use the following instance of the preceding definition: the categories \mathcal{C} and \mathcal{C}' are instantiated by Set^T and $\text{Set}^{T'}$, respectively, for sets T and T' . The functor G is the retyping functor (cf. [Rem. 2.23](#)) associated to some translation of types $g : T \rightarrow T'$. Similarly, the categories \mathcal{D} and \mathcal{D}' are instantiated by Pre^T and $\text{Pre}^{T'}$, and the functor F by

$$F := \Delta^T : \text{Set}^T \rightarrow \text{Pre}^T,$$

and similar for F' :

$$\begin{array}{ccc} \text{Set}^T & \xrightarrow{\Delta^T} & \text{Pre}^T \\ \vec{g} \downarrow & \text{Id} \not\parallel & \downarrow \vec{g} \\ \text{Set}^{T'} & \xrightarrow{\Delta^{T'}} & \text{Pre}^{T'}. \end{array}$$

Given a monad P on $F : \mathcal{C} \rightarrow \mathcal{D}$, the notion of *module over P* generalizes the notion of monadic substitution:

2.90 Definition (Module over a Relative Monad, [Code 6.19](#)): Let $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ be a relative monad and let \mathcal{E} be a category. A *module M over P with codomain \mathcal{E}* is given by

- a map $M : \mathcal{C} \rightarrow \mathcal{E}$ on the objects of the categories involved and
- for all objects c, d of \mathcal{C} , a map

$$\varsigma_{c,d} : \mathcal{D}(Fc, Pd) \rightarrow \mathcal{E}(Mc, Md)$$

such that the following diagrams commute for all suitable morphisms f and g :

$$\begin{array}{ccc} Mc & \xrightarrow{\varsigma(f)} & Md \\ & \searrow \varsigma(\sigma(g) \circ f) & \downarrow \varsigma(g) \\ & & Me \end{array} \quad \begin{array}{ccc} Mc & & \\ & \searrow \text{id} & \searrow \varsigma(\eta_c) \\ & & Mc. \end{array}$$

A functoriality (rmlift) for such a module M is then defined similarly to that for monads: for any morphism $f : c \rightarrow d$ in \mathcal{C} we set

$$M(f) := \text{rmlift}_M(f) := \varsigma(\eta \circ Ff) .$$

The following examples of modules are instances of constructions explained in the next section:

2.91 Example (Ex. 2.85 cont.): The map $\text{ULC}_\beta : V \mapsto \text{ULC}_\beta(V)$ yields a module over the relative monad ULC_β , the *tautological module* ULC_β .

2.92 Example: Recall that $V' := V + 1$. The map $\text{ULC}'_\beta : V \mapsto \text{ULC}_\beta(V')$ inherits the structure of an ULC_β -module from the tautological module ULC_β (cf. Ex. 2.91). We call ULC'_β the *derived module* of the module ULC_β ; cf. also Sect. 2.4.2.

2.93 Example: The map $V \mapsto \text{ULC}_\beta(V) \times \text{ULC}_\beta(V)$ inherits a structure of an ULC_β -module from the tautological module ULC_β .

A *module morphism* is a family of morphisms that is compatible with module substitution in the source and target modules:

2.94 Definition (Morphism of Relative Modules, Code 6.20): Let M and N be two relative modules over $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ with codomain \mathcal{E} . A *morphism of relative P -modules* from M to N is given by a collection of morphisms $\rho_c \in \mathcal{E}(Mc, Nc)$ such that for all morphisms $f \in \mathcal{D}(Fc, Pd)$ the following diagram commutes:

$$\begin{array}{ccc} Mc & \xrightarrow{\varsigma^M(f)} & Md \\ \rho_c \downarrow & & \downarrow \rho_d \\ Nc & \xrightarrow{\varsigma^N(f)} & Nd. \end{array}$$

The modules over P with codomain \mathcal{E} and morphisms between them form a category called $\text{RMod}(P, \mathcal{E})$ (in the digital library: $\text{RMOD } P \text{ E}$). Composition and identity morphisms of modules are defined by pointwise composition and identity, similarly to the category of monads.

2.95 Example (Ex. 2.91, 2.92, Ex. 2.93 cont.): Abstraction and application are morphisms of ULC_β -modules:

$$\begin{aligned} \text{Abs} : \text{ULC}'_\beta &\rightarrow \text{ULC}_\beta , \\ \text{App} : \text{ULC}_\beta \times \text{ULC}_\beta &\rightarrow \text{ULC}_\beta . \end{aligned}$$

2.4.2. Constructions on Relative Monads and Modules

The following constructions are analogous to those of Sect. 2.2.2.

2.96 Definition (Tautological Module): Every monad P on $F : \mathcal{C} \rightarrow \mathcal{D}$ yields a module (P, σ^P) — also denoted by P — over itself, i.e. an object in the category $\text{RMod}(P, \mathcal{D})$.

2.97 Definition (Constant and Terminal Module): Let P be a monad on $F : \mathcal{C} \rightarrow \mathcal{D}$. For any object $e \in \mathcal{E}$ the constant map $T_e : \mathcal{C} \rightarrow \mathcal{E}, c \mapsto e$ for all $c \in \mathcal{C}$, is equipped with the structure of a P -module by setting $\varsigma_{c,d}(f) = \text{id}_e$. In particular, if \mathcal{E} has a terminal object $1_{\mathcal{E}}$, then the constant module $T_{1_{\mathcal{E}}} : c \mapsto 1_{\mathcal{E}}$ is terminal in $\text{RMod}(P, \mathcal{E})$.

2.98 Definition (Postcomposition with a functor): Let P be a monad on $F : \mathcal{C} \rightarrow \mathcal{D}$, and let M be a P -module with codomain \mathcal{E} . Let $G : \mathcal{E} \rightarrow \mathcal{X}$ be a functor. Then the object map $G \circ M : \mathcal{C} \rightarrow \mathcal{X}$ defined by $c \mapsto G(M(c))$ is equipped with a P -module structure by setting, for $c, d \in \mathcal{C}$ and $f \in \mathcal{D}(Fc, Pd)$,

$$\varsigma^{G \circ M}(f) := G(\varsigma^M(f)) .$$

For $M := P$ and G a constant functor mapping to an object $x \in \mathcal{X}$ and its identity morphism id_x , we obtain the constant module (T_x, id) as in the preceding definition.

2.99 Definition (Pullback Module): Suppose given two relative monads P and Q and a morphism $\tau : P \rightarrow Q$ as in Def. 2.87. Let N a Q -module with codomain \mathcal{E} . We define a P -module h^*M to \mathcal{E} with object map

$$c \mapsto M(Gc)$$

by defining the substitution map, for $f : Fc \rightarrow Pd$, as

$$\varsigma^{h^*M}(f) := \varsigma^M(h_d \circ G'f \circ N_c) .$$

The module thus defined is called the *pullback module of N along h* . The pullback extends to module morphisms and is functorial.

2.100 Definition (Induced Module Morphism): With the same notation as before, the monad morphism h induces a morphism of P -modules $h : G'P \rightarrow h^*Q$. Note that the domain module is the module obtained by postcomposing P with G' , whereas for (plain) monads the module was just the tautological module of the domain monad.

2.101 Definition (Product): Suppose the category \mathcal{E} is equipped with a product. Let M and N be P -modules with codomain \mathcal{E} . Then the map

$$M \times N : \mathcal{C} \rightarrow \mathcal{E}, \quad c \mapsto Mc \times Nc$$

is canonically equipped with a substitution and thus constitutes a module called the *product of M and N* . This construction extends to a product on $\text{RMod}(P, \mathcal{E})$.

2.4.3. Derivation & Fibre

We are particularly interested in monads on the functor $\Delta^T : \text{Set}^T \rightarrow \text{Pre}^T$ for some set T , and modules over such monads. The constructions on modules over monads of Sect. 2.2.3, derivation (cf. Sect. 2.2.3.1) and fibre modules (cf. Sect. 2.2.3.2), carry over to modules over monads on Δ^T .

2.102 Definition: Given a monad P over Δ^T and a P -module M with codomain \mathcal{E} , we define the derived module of M with respect to $u \in T$ by setting

$$M^u(V) := M(V^{*u}) .$$

The module substitution is defined, for $f \in \text{Pre}^T(\Delta^T V, PW)$, by

$$\varsigma^{M^u}(f) := \varsigma^M({}_uf) .$$

Here the “shifted” map

$${}_uf \in \text{Pre}^T(\Delta^T(V^{*u}), P(W^{*u}))$$

is the adjunct under the adjunction of Rem. 2.21 of the coproduct map

$$\varphi({}_uf) := [P(\text{inl}) \circ f, \eta(\text{inr}(*))] : V^{*u} \rightarrow UP(W^{*u}) ,$$

where $[\text{inl}, \text{inr}] = \text{id} : W^{*u} \rightarrow W^{*u}$. Derivation is an endofunctor on the category of P -modules with codomain \mathcal{E} .

2.103 Notation: In case the set T of types is $T = \{*\}$ the singleton set of types, i.e. when talking about untyped syntax, we denote by M' the derived module of M . Given a natural number n , we denote by M^n the module obtained by deriving n times the module M .

Analogously to Sect. 2.2.3, we derive more generally with respect to a natural transformation $\tau : 1 \rightarrow \mathcal{T}U_n$ as in Def. 2.57:

2. Category–Theoretic Constructions

2.104 Definition (Derived Module): Let $\tau : 1 \rightarrow \mathcal{T}U_n$ be a natural transformation. Let T be a set and P be a relative monad on Δ_n^T . Given any P –module M , we call *derivation of M with respect to τ* the module with object map $M^\tau(V) := M(V^{\tau(V)})$.

2.105 Definition: Let P be a relative monad over F , and M a P –module with codomain \mathcal{E}^T for some category \mathcal{E} . The *fibre module* $[M]_t$ of M with respect to $t \in T$ has object map

$$c \mapsto M(c)(t) = M(c)_t$$

and substitution map

$$\varsigma^{[M]_t}(f) := (\varsigma^M(f))_t .$$

This definition generalizes to fibres with respect to a natural transformation as in [Def. 2.104](#).

The pullback operation commutes with products, derivations and fibres :

2.106 Lemma: Let \mathcal{C} and \mathcal{D} be categories and \mathcal{E} be a category with products. Let $P : \mathcal{C} \rightarrow \mathcal{D}$ and $Q : \mathcal{C} \rightarrow \mathcal{D}$ be monads over $F : \mathcal{C} \rightarrow \mathcal{D}$ and $F' : \mathcal{C}' \rightarrow \mathcal{D}'$, resp., and $\rho : P \rightarrow Q$ a monad morphism. Let M and N be P –modules with codomain \mathcal{E} . The pullback functor is cartesian:

$$\rho^*(M \times N) \cong \rho^*M \times \rho^*N .$$

2.107 Lemma: Consider the setting as in the preceding lemma, with $F = \Delta^T$, and $t \in T$. Then we have

$$\rho^*(M^t) \cong (\rho^*M)^t .$$

2.108 Lemma: Suppose N is a Q –module with codomain \mathcal{E}^T , and $t \in T$. Then

$$\rho^*[M]_t \cong [\rho^*M]_t .$$

2.109 Definition: Recall that the category \mathbf{wPre} is the category of preordered sets and set–theoretic maps (not necessarily monotone) between them ([Def. 2.9](#)). Given a relative monad P on some functor F and a P –module M with codomain \mathbf{Pre} , we can consider M as a P –module with codomain \mathbf{wPre} . We denote this module by \hat{M} . In other words, we have a functor

$$\hat{_} : \mathbf{RMod}(P, \mathbf{Pre}) \rightarrow \mathbf{RMod}(P, \mathbf{wPre})$$

obtained by postcomposition with the forgetful functor from \mathbf{Pre} to \mathbf{wPre} .

2.110 Definition (Substitution of one Variable): Let P be a monad over Δ . For any set X , we define a binary substitution operation

$$\begin{aligned} \text{subst}(X) : P(X^*) \times P(X) &\rightarrow P(X), \\ (y, z) &\mapsto y[* := z] := \sigma(\text{default}(\eta_X, z))(y) , \end{aligned}$$

where “default” is a coproduct map; for $f : A \rightarrow B$ and $z \in B$,

$$\text{default}(f, z) := [f, x \mapsto z] : A + \{*\} \rightarrow B .$$

This defines a morphism of P -modules with codomain $w\text{Pre}$,

$$\text{subst}^P : \hat{P}' \times \hat{P} \rightarrow \hat{P} .$$

The reason why we have to consider the category $w\text{Pre}$ with *all set-theoretic* maps instead of just monotone maps is that subst^P is not necessarily monotone in its second argument, cf. [Rem. 2.86](#).

The untyped substitution of [Def. 2.110](#) actually is a special case of the following typed substitution:

2.111 Definition (Substitution of *one* Variable, typed): Let T be a (nonempty) set and let P be a monad over Δ^T . For any $s, t \in T$ and $X \in \text{Set}^T$ we define a binary substitution operation

$$\begin{aligned} \text{subst}_{s,t}(X) : P(X^{*s})_t \times P(X)_s &\rightarrow P(X)_t, \\ (y, z) &\mapsto y[* := z] := \sigma(\text{default}(\eta_X, z))(y) . \end{aligned}$$

For any pair $(s, t) \in T^2$, we thus obtain a morphism of P -modules

$$\text{subst}_{s,t}^P : [\hat{P}^s]_t \times [\hat{P}]_s \rightarrow [\hat{P}]_t .$$

3. SIMPLE TYPE SYSTEMS

In this chapter we present two generalizations to simple type systems of Hirschowitz and Maggesi’s initiality theorem for untyped syntax [HM07a]:

- in Sect. 3.2 we review Zsidó’s theorem [Zsi10, Chapt. 6].
- In Sect. 3.3 we prove a variant of Zsidó’s theorem which accounts for translations between languages over *different* sets of object types.

We explain the difference between the two abovementioned theorems in more detail:

in Zsidó’s theorem, the underlying set of types of a signature — and thus of the term language the signature specifies — is given as a fixed parameter. In particular, all the models — representations — of the signature have the same underlying set of types. Furthermore, this set does not necessarily have inductive structure, as opposed to the sets of types we characterize via initiality in Sect. 3.1 — the content of Sect. 3.2 is independent of that of Sect. 3.1.

In our variant of Zsidó’s theorem we prove in Sect. 3.3, a language is specified by a pair (S, Σ) of signatures, a signature S for *types* as presented in Sect. 3.1, and a signature Σ for *terms* over the signature S . A representation of such a signature is given by a pair of a representation of S and a representation of Σ . In particular, we consider models of (S, Σ) whose underlying set of types is different from the set freely generated by the signature S . The initiality result of Sect. 3.3 thus characterizes both the types and terms freely generated by a signature as initial object in a category of representations.

As running examples, we consider the simply-typed lambda calculus and Plotkin’s PCF [Plo77]. In Sect. 3.4 we present a logic translation from classical to intuitionistic propositional logic as an instance of our theorem of Sect. 3.3. Before focusing on *term signatures*, however, we review, in Sect. 3.1, *algebraic* signatures as treated by Birkhoff [Bir35]. Algebraic signatures are used in Sect. 3.3 for the specification of the set of types of a language.

3.1. Signatures for Types

We present *algebraic signatures*, which later are used to specify the *object types* of the languages we consider. Algebraic signatures and their models were first considered by Birkhoff [Bir35].

3. Simple Type Systems

3.1 Definition (Algebraic Signature): An *algebraic signature* S is a family of natural numbers, i.e. a set J_S and a map (carrying the same name as the signature) $S : J_S \rightarrow \mathbb{N}$. For $j \in J_S$ and $n \in \mathbb{N}$, we also write $j : n$ instead of $j \mapsto n$. An element of J resp. its image under S is called an *arity* of S .

3.2 Example (Algebraic Signature of Ex. 1.3): The algebraic signature of the types of the simply-typed lambda calculus is given by

$$S_{\text{TLC}} := \{* : 0, (\rightsquigarrow) : 2\}.$$

To any algebraic signature we associate a category of *representations*. We call *representation of S* any set U equipped with operations according to the signature S . A *morphism of representations* is a map between the underlying sets that is compatible with the operations on either side in a suitable sense. Representations and their morphisms form a category. We give the formal definitions:

3.3 Definition (Representation of an Algebraic Signature S , S -Algebra): A *representation* R of an algebraic signature S — also known as *S -algebra* — is given by

- a set X and
- for each $j \in J_S$, an operation $j^R : X^{S(j)} \rightarrow X$.

In the following, given a representation R , we write R also for its underlying set.

3.4 Example: The language PCF [Plo77, HO00] (see also Sect. A.1) is a simply-typed lambda calculus with a fixed point operator and arithmetic constants. Let $J := \{\iota, o, (\Rightarrow)\}$. The signature of the types of PCF is given by the arities

$$S_{\text{PCF}} := \{\iota : 0, o : 0, (\Rightarrow) : 2\}.$$

A representation T of S_{PCF} is given by a set T and three operations,

$$\iota^T : T, o^T : T, (\Rightarrow)^T : T \times T \rightarrow T.$$

A morphism of representations is given by a map between the underlying sets that is compatible with the representation structure:

3.5 Definition (Morphisms of Representations): Given two representations T and U of the algebraic signature S , a *morphism* from T to U is a map $f : T \rightarrow U$ such that, for any arity $n = S(j)$ of S , we have

$$f \circ j^T = j^U \circ \underbrace{(f \times \dots \times f)}_{n \text{ times}}.$$

3.6 Example (Ex. 3.4 continued): Given two representations T and U of S_{PCF} , a morphism from T to U is a map $f : T \rightarrow U$ between the underlying sets such that, for any $s, t \in T$,

$$\begin{aligned} f(\iota^T) &= \iota^U, \\ f(o^T) &= o^U \quad \text{and} \\ f(s \Rightarrow^T t) &= f(s) \Rightarrow^U f(t). \end{aligned}$$

Representations of an algebraic signature S and their morphisms form a category.

3.7 Lemma: *Let (J, S) (or S for short) be an algebraic signature. The category of representations of S has an initial object \hat{S} .*

Proof. We cut the proof into small steps:

- In a type-theoretic setting the set — also called \hat{S} — which underlies the initial representation \hat{S} is defined as an inductive set with a family of constructors indexed by J_S :

$$\hat{S} ::= C : \forall j \in J, \hat{S}^{S(j)} \rightarrow \hat{S}.$$

That is, for each arity $j \in J$, we have a constructor $C_j : \hat{S}^{S(j)} \rightarrow \hat{S}$.

- For each arity $j \in J$, we must specify an operation $j^{\hat{S}} : \hat{S}^{S(j)} \rightarrow \hat{S}$. We set

$$j^{\hat{S}} := C_j : \hat{S}^{S(j)} \rightarrow \hat{S},$$

that is, the representation $j^{\hat{S}}$ of an arity $n = S(j)$ is given precisely by its corresponding constructor.

- Given any representation R of S , we specify a map $i_R : \hat{S} \rightarrow R$ between the underlying sets by structural recursion:

$$i_R : \hat{S} \rightarrow R, \quad i_R(C_j(a)) := j^R((i_R)^{S(j)}(a)),$$

for $a \in \hat{S}^{S(j)}$. That is, the image of a constructor function C_j maps recursively on the image of the corresponding representation j^R of R .

- We must prove that i_R is a morphism of representations, that is, that for any $j \in J$ with $S(j) = n$,

$$i_R \circ j^{\hat{S}} = j^R \circ (i_R)^n.$$

Replacing $j^{\hat{S}}$ by its definition yields that this equation is precisely the specification of i_R , see above.

3. Simple Type Systems

- It is the diagram of [Def. 3.5](#) which ensures uniqueness of i_R ; since any morphism of representations $i' : \hat{S} \rightarrow R$ must make it commute, one can show by structural induction that $i' = i_R$. More precisely:

$$\begin{aligned} i'(C_j(a)) &= i'(C_j(a_1, \dots, a_{S(j)})) = j^R(i'(a_1), \dots, i'(a_{S(j)})) \stackrel{i'(a_k) = i_R(a_k)}{=} \\ &= j^R(i_R(a_1), \dots, i_R(a_{S(j)})) = i_R(C_j(a)) . \end{aligned}$$

□

3.8 Example ([Ex. 3.4](#) continued): The set T_{PCF} underlying the initial representation of the algebraic signature S_{PCF} is given by

$$T_{\text{PCF}} ::= \iota \mid o \mid T_{\text{PCF}} \Rightarrow T_{\text{PCF}} .$$

For any other representation R of S_{PCF} the initial morphism $i_R : T_{\text{PCF}} \rightarrow R$ is given by the clauses

$$\begin{aligned} i_R(\iota) &= \iota^R \\ i_R(o) &= o^R \\ i_R(s \Rightarrow t) &= i_R(s) \Rightarrow^R i_R(t) . \end{aligned}$$

3.2. Zsidó's Theorem Reviewed

We present Zsidó's initiality theorem [[Zsi10](#), Chapt. 6] (cf. [Thm. 3.28](#)) for simply-typed abstract syntax. Its formalization in the proof assistant Coq is explained in [Chapt. 7](#). Throughout this section the number given in the name of each definition points to the implementation of this definition in Coq. For instance, the implementation of Simple Monad Morphisms ([Def. 3.12](#)) is given in [Code 6.13](#).

Our presentation follows the pattern outlined at the beginning of [Sect. 1.2](#): in [Sect. 3.2.1](#) we present classic signatures in two different ways. Afterwards, in [Sect. 3.2.2](#), we give the definition of representations of such signatures. Finally, in [Sect. 3.2.3](#), we state the main theorem, proved by Zsidó [[Zsi10](#)].

3.2.1. Signatures for Terms

In [Sect. 3.2.1.1](#) we give a purely syntactical definition of *classic* arities. Afterwards, in [Sect. 3.2.1.2](#) we give a definition of arities as pairs of functors on suitable categories, and identify a subclass of arities which are in one-to-one correspondence with classic arities. We thus call arities of this subclass *classic* as well. In the following we fix a set T of object types.

3.2.1.1. Arities, syntactically

Syntactically, a classic arity consists of an element of $t_0 \in T$ which specifies the output type of a constructor, as well as a list of pairs $([t_{i,1}, \dots, t_{i,m_i}], t_i)$, where $t_{i,k}, t_i \in T$. Each such pair represents an argument of the corresponding constructor: the element t_i denotes the object type of the argument, whereas the list $[t_{i,1}, \dots, t_{i,m_i}]$ specifies the types of the variables that are bound by the constructor in this argument.

3.9 Definition (Classic T -Arity, T -Signature): A classic arity is of the form

$$[([t_{1,1}, \dots, t_{1,m_1}], t_1), \dots, ([t_{n,1}, \dots, t_{n,m_n}], t_n)] \rightarrow t_0 ,$$

where $t_{i,k}$ and t_i are elements of T . We use an arrow to separate the data specifying input data and output data, respectively. A *signature* is a family of arities. For a formalized definition, see the Coq code snippets [Code 7.1](#) and [Code 7.2](#).

3.10 Example (Signature of TLC): The signature of the simply-typed lambda calculus (cf. [Ex. 1.3](#)) is given by

$$\{\text{abs}_{s,t} : [([s], t)] \rightarrow (s \rightsquigarrow t) , \quad \text{app}_{s,t} : [([], s \rightsquigarrow t), ([], s)] \rightarrow t\}_{s,t \in T_{\text{TLC}}} .$$

See the code snippet [Code 7.3](#) for a Coq implementation of this example.

3.2.1.2. Arities, semantically

In this section we give a definition of arities as pairs of functors between suitable categories. The source category (cf. [Def. 3.13](#)) is a category of monads and morphisms of monads, whereas the target category (cf. [Def. 3.15](#)) mixes modules over different such monads.

At first, in [Rem. 3.11](#), we present an alternative characterization of algebraic arities. This alternative point of view is then adapted to allow for the specification of arities for terms.

3.11 Remark *Algebraic Arities viewed differently*: An algebraic arity $j : n$ as presented in [Sect. 3.1](#) associates, to any set X , the set $\text{dom}(j, X) := X^n$, the *domain* set. A representation R of this arity j in a set X then is given by a map $j^R : X^n \rightarrow X$. More formally, the domain set is given via a functor $\text{dom}(j) : \text{Set} \rightarrow \text{Set}$ which associates to any set X the set X^n . Similarly, we might also speak of a *codomain* functor for any arity, which — for algebraic arities — is given by the identity functor. A representation R of j in a set X then is given by a morphism

$$j^R : \text{dom}(j)(X) \rightarrow \text{cod}(j)(X) .$$

We take the perspective of [Rem. 3.11](#) in order to define arities and signatures for *terms*: given a set T of object types, an arity α for terms typed over T is a pair of functors

3. Simple Type Systems

$(\text{dom}(\alpha), \text{cod}(\alpha))$ associating two P -modules $\text{dom}(\alpha)(P)$ and $\text{cod}(\alpha)(P)$, to any suitable monad P . A suitable monad here is a monad P on the category Set^T . A representation R of α in a such a monad P is a module morphism

$$\alpha^R : \text{dom}(\alpha)(P) \rightarrow \text{cod}(\alpha)(P) .$$

We consider monads as in [Def. 2.33](#) (also: [Def. 2.65](#)) over a category of the form Set^T for some fixed set T . Throughout this section, morphisms between two such monads over the same category are given by colax monad morphisms *over the identity functor*, i.e. those morphisms of [Def. 2.38](#) (alt. [Def. 2.69](#)) with $F = \text{Id}_{\text{Set}^T}$. For convenience, and as a reference for the implementation in Coq, we explicitly state the definition of these “simple” monad morphisms, using the definition through Kleisli operation (cf. [Def. 2.69](#)) of monads and morphisms:

3.12 Definition (Simple Monad Morphism, [Code 6.13](#)): Let P and Q be two monads over a category \mathcal{C} . A *simple morphism of monads* τ from P to Q is given by a collection of morphisms $\tau_c \in \mathcal{C}(Pc, Qc)$ such that the following diagrams commute for all suitable morphisms f :

$$\begin{array}{ccc} Pc & \xrightarrow{\sigma^P(f)} & Pd \\ \tau_c \downarrow & & \downarrow \tau_d \\ Qc & \xrightarrow{\sigma^Q(\tau_d \circ f)} & Qd, \end{array} \quad \begin{array}{ccc} c & \xrightarrow{\eta_c^P} & Pc \\ & \searrow \eta_c^Q & \downarrow \tau_c \\ & & Qc. \end{array}$$

3.13 Definition (Category $\text{Mon}(\mathcal{C})$ of Monads on \mathcal{C}): Given a category \mathcal{C} , we define the category $\text{Mon}(\mathcal{C})$ to be the category whose objects are monads over \mathcal{C} . A morphism from P to Q in this category is a monad morphism as in [Def. 3.12](#). We denote by $I_{\mathcal{C}} : \text{Mon}(\mathcal{C}) \rightarrow \mathbf{Mnd}_{\text{colax}}$ the inclusion functor.

We define a category in which modules over different monads — but with the same codomain category — are mixed together. This category can be defined as a particular *colax comma category*. However, we also give an explicit description of the objects and morphisms of this category.

3.14 Definition (Colax Comma Category): Let \mathcal{C} be a 2-category, and $c \in \mathcal{C}$ be an object of \mathcal{C} . Let \mathcal{A} be a category and let $F : \mathcal{A} \rightarrow \mathcal{C}$ be a functor. An object of the *colax comma category* $(F \downarrow c)$ is given by a pair $(a, f : Fa \rightarrow c)$ of an object $a \in \mathcal{A}$ and a morphism $f : a \rightarrow c$. A morphism to another such $(b, g : Fb \rightarrow c)$ is given by a pair $(h : a \rightarrow b, \alpha)$ as in the diagram

$$\begin{array}{ccc} Fa & \xrightarrow{f} & c \\ & \Downarrow \alpha & \\ Fa & \xrightarrow{Fh} Fb & \xrightarrow{g} c \end{array} .$$

While the above definition is not the most general definition possible for a colax comma category, it is sufficient for our needs:

3.15 Definition (Large Category $\mathbf{LMod}(\mathcal{C}, \mathcal{D})$ of Modules): Given two categories \mathcal{C} and \mathcal{D} , we define the category $\mathbf{LMod}(\mathcal{C}, \mathcal{D})$ to be the colax comma category $(I_{\mathcal{C}} \downarrow \text{Id}_{\mathcal{D}})$. An object of this category is a monad P over \mathcal{C} together with a P -module with codomain \mathcal{D} (cf. Def. 2.43). A morphism (f, h) to another such (Q, N) is given by a morphism $f : P \rightarrow Q$ of monads over the identity functor — i.e., a morphism in $\mathbf{Mon}(\mathcal{C})$ — and a morphism of modules $h : M \rightarrow f^*N = N \circ f$:

$$\begin{array}{ccc} & M & \\ \curvearrowright & \Downarrow h & \curvearrowleft \\ P & & \text{Id}_{\mathcal{D}} \\ \curvearrowleft & N \circ f & \end{array} .$$

3.16 Definition (Tautological Module): To any monad $R \in \mathbf{Mon}(\mathcal{C})$ we associate the tautological module $\Theta(R)$ of R ,

$$\Theta(R) := (R, R) \in \mathbf{LMod}(\mathcal{C}, \mathcal{C}) .$$

This construction extends to a functor $\Theta : \mathbf{Mon}(\mathcal{C}) \rightarrow \mathbf{LMod}(\mathcal{C}, \mathcal{C})$.

A half-arity associates a P -module towards \mathbf{Set} to any monad P over \mathbf{Set}^T :

3.17 Definition (Half-Arity): A *half-arity over T* is a functor

$$a : \mathbf{Mon}(\mathbf{Set}^T) \rightarrow \mathbf{LMod}(\mathbf{Set}^T, \mathbf{Set})$$

from the category of monads over \mathbf{Set}^T to the large category of modules over such monads with codomain \mathbf{Set} , such that

$$\pi_1 \circ a = \text{id}_{\mathbf{Mon}(\mathbf{Set}^T)} . \quad (3.2.1)$$

This last condition given in Disp. (3.2.1) ensures that each monad maps to a module *over itself*. For a monad $R \in \mathbf{Mon}(\mathbf{Set}^T)$, we thus sometimes omit the first component R of the image $a(R)$ and consider $a(R) \in \mathbf{Mod}(R, \mathbf{Set})$.

3.18 Definition (Arity, Signature): A T -arity s is a pair $(\text{dom}(s), \text{cod}(s))$ of half-arithies over T ,

$$\text{dom}(s), \text{cod}(s) : \mathbf{Mon}(\mathbf{Set}^T) \rightarrow \mathbf{LMod}(\mathbf{Set}^T, \mathbf{Set}) ,$$

written $\text{dom}(s) \rightarrow \text{cod}(s)$. A T -signature is a family of T -arithies.

We give some important examples of half-arithies over the set T . Note that, by the convention of Def. 3.17, we omit the first component of objects of the large category of modules $\mathbf{LMod}(\mathbf{Set}^T, \mathbf{Set})$.

3. Simple Type Systems

3.19 Definition: Let T be a nonempty set, and let $t \in T$ be an element of T .

- The map $[\Theta]_t : \text{Mon}(\text{Set}^T) \rightarrow \text{LMod}(\text{Set}^T, \text{Set})$ with object map $R \mapsto (R, [R]_t)$ is a half-arity — the *fibre with respect to t* — over T .
- If M is a half-arity over T , so is $M^t : \text{Mon}(\text{Set}^T) \rightarrow \text{LMod}(\text{Set}^T, \text{Set})$, $M^t(R) := M(R)^t$ (cf. [Def. 2.55](#)). By iterating, given $t_1, \dots, t_n \in T$, the functor

$$M^{(t_1, \dots, t_n)} : R \mapsto (\dots (M(R)^{t_1}) \dots)^{t_n}$$

is a half-arity.

- If M and N are half-arithies over T , then so is the product $M \times N : \text{Mon}(\text{Set}^T) \rightarrow \text{LMod}(\text{Set}^T, \text{Set})$:

$$M \times N : R \mapsto M(R) \times N(R) .$$

- The map $R \mapsto *$, where $* : V \mapsto 1_{\text{Set}}$ is the terminal object in $\text{Mod}(R, \text{Set})$, is a half-arity over T .

An *arity* is a pair of half-arithies. We are only interested in *classic* arities, whose domain and codomain functors are of a specific form:

3.20 Definition (Classic T -Arity, T -Signature (II)): We call *classic T -arity* any T -arity s of the form

$$s = [\Theta]_{t_1}^{t_{1,1} \dots t_{1,m_1}} \times \dots \times [\Theta]_{t_n}^{t_{n,1} \dots t_{n,m_n}} \rightarrow [\Theta]_{t_0} \quad (3.2.2)$$

for $t_{i,j}, t_i \in T$. A *classic T -signature* is a collection of such classic arities.

To an operator that binds m_k variables of types $t_{k,1}, \dots, t_{k,m_k}$ in its k -th argument of type t_k , and which yields a term of type t_0 , we associate the arity given in [Disp. \(3.2.2\)](#).

3.21 Remark: The classic T -arithies and T -signatures of [Def. 3.20](#) and of [Def. 3.9](#) are in bijection, respectively. We can thus specify T -signatures by simply giving a term of the simple data type defined in [Def. 3.9](#). In the Coq formalization, arities and signatures are defined via such data types, cf. [Code 7.1](#) and [Code 7.2](#).

3.22 Remark: In [Def. 3.20](#) we can have $n = 0$, yielding an arity for constants of, say, object type $t_0 \in T$,

$$s = * \rightarrow [\Theta]_{t_0} .$$

Such an arity then is given by an empty list of arguments according to [Def. 3.9](#). An example of a constant arity is given in [Ex. 3.48](#).

As an example we discuss the classic signature of the simply typed lambda calculus:

3.23 Example (Signature of TLC, [Code 7.3](#)): Consider the example of the simply-typed lambda calculus (cf. [Exs. 1.3, 2.37](#)). Its signature is given syntactically in [Ex. 3.10](#). Equivalently, it is given by the signature

$$\Sigma_{\text{TLC}} = \{\text{abs}_{s,t}, \text{app}_{s,t}\}_{s,t \in T_{\text{TLC}}}$$

with

$$\begin{aligned} \text{abs}_{s,t} &:= [\Theta]_t^s \rightarrow [\Theta]_{s \rightsquigarrow t} \quad \text{and} \\ \text{app}_{s,t} &:= [\Theta]_{s \rightsquigarrow t} \times [\Theta]_s \rightarrow [\Theta]_t. \end{aligned}$$

3.24 Remark: Note that in [Ex. 3.23](#) we do not need to explicitly specify an arity for the Var term constructor in order to obtain the simply-typed lambda calculus as presented in [Ex. 1.3](#). Indeed, by building models from monads (cf. [Def. 3.25](#)) every model is by definition equipped with a corresponding operation — the unit of the underlying monad.

3.2.2. Representations

A representation of an arity s in a monad P is given by a morphism of P -modules whose domain and codomain are determined by s :

3.25 Definition (Representation of a T -Signature, [Code 7.4](#)): A representation R of a T -signature Σ is given by

- a monad P on the category Set^T and
- for any arity $s \in \Sigma$, a morphism of modules in $\text{LMod}(\text{Set}^T, \text{Set})$,

$$s^R : \text{dom}(s, P) \rightarrow \text{cod}(s, P),$$

$$\text{such that } \pi_1(s^R) = \text{id}_P.$$

Given a representation R , we denote by R also the underlying monad.

Morphisms of representations are monad morphisms that are compatible with the representation module morphisms:

3.26 Definition (Morphism of Representations): Let P and Q be representations of a T -signature Σ . A *morphism of representations* $f : P \rightarrow Q$ is a morphism f between the underlying monads such that the following diagram commutes for any arity s of Σ :

$$\begin{array}{ccc} \text{dom}(s, P) & \xrightarrow{s^P} & \text{cod}(s, P) \\ \text{dom}(s, f) \downarrow & & \downarrow \text{cod}(s, f) \\ \text{dom}(s, Q) & \xrightarrow{s^Q} & \text{cod}(s, Q). \end{array} \quad (3.2.3)$$

3. Simple Type Systems

The preceding diagram can be seen as a diagram in two different categories, either in the category $\text{LMod}(\text{Set}^T, \text{Set})$, or in the category $\text{Mod}(P, \text{Set})$ of P -modules.

3.27 Definition (Category of Representations): Morphisms of representations can be composed: the composition of the underlying monad morphisms again gives a morphism of representations. Similarly the identity morphism of monads is a morphism of representations. Two morphisms of representations are said to be equal if their underlying morphisms of monads are equal. Representations and their morphisms of a signature Σ form a category $\text{Rep}(\Sigma)$.

3.2.3. Initiality

The main theorem states that any T -signature admits an initial representation:

3.28 Theorem: *Let Σ be a classic T -signature. Then the category $\text{Rep}(\Sigma)$ of representations of Σ has an initial object.*

3.29 Remark: The monad underlying the initial representation associates, to any context $V \in \text{Set}^T$, the set of terms of the syntax of Σ with free variables in V . The module morphisms of the initial representation are given by the constructors of this syntax.

A set-theoretic construction of the syntax as well as a proof of the theorem is given in Zsidó's PhD thesis [Zsi10]. In Sect. 7.2 we explain the implementation of the main theorem in a type-theoretic setting in the proof assistant Coq.

3.3. Extending Zsidó's Theorem to Varying Types

Zsidó's initiality result of Thm. 3.28 does not account for varying object sorts. Indeed, given a signature Σ over a set T of object sorts, any representation of Σ “has” the same set of sorts T , i.e. its underlying monad is a monad on the category Set^T . In this section we give a new definition of signatures and their representations, and prove that the resulting category of representations has an initial object. The iteration operator obtained from this initiality result accounts for translations between languages over different sets of sorts. We define a *typed signature* to be a pair (S, Σ) consisting of an algebraic signature S for sorts, and a signature Σ for terms typed over the sorts specified by S . A representation of such a typed signature consists of a representation of the sort signature S in some set T and a representation of Σ in a monad over the category Set^T . Translations of sorts are given by morphisms of representations of S , that is, by maps of sets that are compatible with the representations of sorts constructors in the source and target. Compared to Zsidó, we thus restrict ourselves to sets of sorts that have *inductive structure*, whereas for Zsidó, the set of sorts is given by an arbitrary parameter.

3.3.1. Signatures for Types & Terms

Before starting with the formal definitions, we informally consider the example of the simply-typed lambda calculus; its signature for terms was given in the preceding section (cf. Ex. 3.23) as:

$$\{\text{abs}_{s,t} := [([s], t)] \rightarrow (s \rightsquigarrow t) \ , \ \text{app}_{s,t} := [([], s \rightsquigarrow t), ([], s)] \rightarrow t\}_{s,t \in T_{\text{TLC}}} \ . \quad (3.3.1)$$

The parameters s and t range over the set T_{TLC} of types, the initial representation of the signature for types from Ex. 3.2. In particular, we have $2 \times T_{\text{TLC}}^2$ arities in this signature.

Our goal is to consider representations of the simply-typed lambda calculus in monads over categories of the form Set^T for any set T — provided that T is equipped with a representation of the signature S_{TLC} . Clearly, the above signature of Disp. (3.3.1), with its strong dependence on the set T_{TLC} is not well-suited to express this. Instead of the above signature, we would like to write

$$\{\text{abs} := [([1], 2)] \rightarrow (1 \rightsquigarrow 2) \ , \ \text{app} := [([], 1 \rightsquigarrow 2), ([], 1)] \rightarrow 2\} \ . \quad (3.3.2)$$

What is the intended meaning of such a signature? For any representation T of S_{TLC} , the variables 1 and 2 range over elements of T . In this way the number of abstractions and applications depends on the representation T of S_{TLC} : intuitively, a representation of the above signature of Disp. (3.3.2) over a representation T of T_{TLC} has T^2 abstractions and T^2 applications — one for each pair of elements of T . As an example, for the final representation of S_{TLC} in the singleton set, one obtains only one abstraction and one application morphism. We call arities, that contain object type variables, arities of *higher degree*, where the degree of such an arity denotes the number of (distinct) type variables. For instance, the arities abs and app of Disp. (3.3.2) are of degree 2.

3.3.1.1. Term Arities, syntactically

In Sect. 3.2, arities over a fixed set of object types T were defined purely syntactically, namely using pairs and lists, cf. Def. 3.9. We give a similar syntactic characterization of arities over a fixed algebraic signature S for types as in Def. 3.1.

3.30 Definition (Type of Degree n): For $n \geq 1$, we call *types of S of degree n* the elements of the set $S(n)$ of types associated to the signature S with free variables in the set $\{1, \dots, n\}$. We set $S(0) := \hat{S}$. Formally, the set $S(n)$ may be obtained as the initial representation of the signature S enriched by n nullary arities.

Types of degree n are used to form classic arities of degree n :

3.31 Definition (Classic Arity of Degree n): A classic arity for terms over the signature S for types of degree n is of the form

$$[[[t_{1,1}, \dots, t_{1,m_1}], t_1), \dots, ([t_{k,1}, \dots, t_{k,m_k}], t_k)] \rightarrow t_0 \ , \quad (3.3.3)$$

3. Simple Type Systems

where $t_{i,j}, t_i \in S(n)$. More formally, a classic arity of degree n over S is a pair consisting of an element $t_0 \in S(n)$ and a list of pairs. where each pair itself consists of a list $[t_{i,1}, \dots, t_{i,m_i}]$ of elements of $S(n)$ and an element t_i of $S(n)$.

A classic arity of the form given in [Disp. \(3.3.3\)](#) denotes a constructor — or a family of constructors, for $n \geq 1$ — whose output type is t_0 , and whose k inputs are terms of type t_i , respectively, in each of which variables of type according to the list $[t_{i,1}, \dots, t_{i,m_i}]$ are bound by the constructor.

3.32 Remark: For an arity as given in [Disp. \(3.3.3\)](#) we also write

$$[\Theta_n^{t_{1,1}, \dots, t_{1,m_1}}]_{t_1} \times \dots \times [\Theta_n^{t_{k,1}, \dots, t_{k,m_k}}]_{t_k} \rightarrow [\Theta_n]_{t_0} . \quad (3.3.4)$$

Examples of (classic) arities are to be found in [Ex. 3.47](#) and [Sect. 3.4](#).

3.33 Remark *Implicit Degree*: Any arity of degree $n \in \mathbb{N}$ as in [Def. 3.31](#) can also be considered as an arity of degree $n + 1$. We denote by $S(\omega)$ the set of types associated to the type signature S with free variables in \mathbb{N} . Then any arity of degree $n \in \mathbb{N}$ can be considered as an arity built over $S(\omega)$. Conversely, any arity built over $S(\omega)$ only contains a finite set of free variables in \mathbb{N} , and can thus be considered to be an arity of degree n for some $n \in \mathbb{N}$. In particular, by suitable renaming of free variables, there is a *minimal* degree for any arity built over $S(\omega)$. We can thus omit the degree — e.g., the lower inner index n in [Disp. \(3.3.4\)](#) —, and specify any arity as an arity over $S(\omega)$, if we really want to consider this arity to be of minimal degree. Otherwise we must specify the degree explicitly.

3.3.1.2. Term Arities, semantically

We now attach a meaning to the purely syntactically defined arities of [Sect. 3.3.1.1](#). More precisely, we define arities as pairs of functors over suitable categories. Afterwards we restrict ourselves to a specific class of functors, yielding arities which are in one-to-one correspondence to — and thus can be compactly specified via — the syntactically defined classic arities of [Sect. 3.3.1.1](#). Accordingly, we call the restricted class of arities also *classic* arities.

Throughout this section, we fix an algebraic signature S for types. An arity α of degree n for terms over S is a pair of functors $(\text{dom}(\alpha), \text{cod}(\alpha))$ associating two P -modules $\text{dom}(\alpha, P)$ and $\text{cod}(\alpha, P)$, each of degree n , to any suitable monad P . A suitable monad here is a monad P on some category Set^T where the set T is equipped with a representation of S . We call such a monad an *S -monad*. A representation R of α in an S -monad P is a module morphism

$$\alpha^R : \text{dom}(\alpha, P) \rightarrow \text{cod}(\alpha, P) .$$

As we have seen in [Ex. 1.3](#), constructors can in fact be *families of constructors* indexed by type variables. For such a constructor indexed n times, we consider modules of degree n (cf. [Rem. 3.37](#)).

We define a family of categories of monads which will play the role of the category defined in [Def. 3.13](#):

3.34 Definition (S -Monad): Given an algebraic signature S , the 2-category $S\text{-Mnd}$ of S -monads is defined as the 2-category whose objects are pairs (T, P) of a representation T of S and a monad $P : \text{Set}^T \rightarrow \text{Set}^T$. A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S -representations $g : T \rightarrow T'$ and a monad morphism $f : P \rightarrow P'$ over the retyping functor \vec{g} (cf. [Rem. 2.23](#)). Transformations are the transformations of $\mathbf{Mnd}_{\text{colax}}$.

Given $n \in \mathbb{N}$, we write $S\text{-Mnd}_n$ for the 2-category whose objects are pairs (T, P) of a representation T of S and a monad P over Set_n^T . A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S -representations $g : T \rightarrow T'$ and a monad morphism $f : P \rightarrow P'$ over the retyping functor $\vec{g}(n)$ (cf. [Def. 2.28](#)).

We call $I_{S,n} : S\text{-Mnd}_n \rightarrow \mathbf{Mnd}_{\text{colax}}$ the functor which forgets the representation of S .

We define a “large category of modules” in which modules over different S -monads are mixed together:

3.35 Definition (Large Category of Modules): Given a natural number $n \in \mathbb{N}$, an algebraic signature S and a category \mathcal{D} , we call $\text{LMod}_n(S, \mathcal{D})$ the colax comma category $I_{S,n} \downarrow (\mathcal{D}, \text{Id})$. An object of this category is a pair (P, M) of a monad $P \in S\text{-Mnd}_n$ and a P -module with codomain \mathcal{D} . A morphism to another such (Q, N) is a pair (f, h) of an S -monad morphism $f : P \rightarrow Q$ in $S\text{-Mnd}_n$ and a transformation $h : M \rightarrow f^*N$:

$$\begin{array}{ccc} & M & \\ \curvearrowright & \Downarrow h & \curvearrowleft \\ P & & \text{Id}_{\mathcal{D}} \\ \curvearrowleft & N \circ f & \end{array} .$$

A half-arity over S of degree n is given by a functor from the category of monads to the large category of modules:

3.36 Definition (Half-Arity over S (of degree n)): Given an algebraic signature S and $n \in \mathbb{N}$, we call *half-arity over S of degree n* a functor

$$\alpha : S\text{-Mnd} \rightarrow \text{LMod}_n(S, \text{Set})$$

which is pre-inverse to the forgetful functor.

Taking into account [Rem. 3.37](#), this means that a half-arity of degree n associates to any S -monad R — with representation of S in a set T — a family of R -modules indexed n times by T .

3. Simple Type Systems

3.37 Remark *Module of Higher Degree corresponds to a Family of Modules:* Let \mathcal{C} be a category, let T be a set and R be a monad on \mathcal{C}^T . Suppose $n \in \mathbb{N}$, and let \mathcal{D} be a category. Then modules over R_n with codomain \mathcal{D} correspond precisely to families of R -modules indexed by T^n with codomain \mathcal{D} by (un)currying. More precisely, let M be an R_n -module. Given $\mathbf{t} \in T^n$, we define an R -module $M_{\mathbf{t}}$ by

$$M_{\mathbf{t}}(c) := M(c, \mathbf{t}) .$$

Module substitution for $M_{\mathbf{t}}$ is given, for $f \in \mathcal{C}^T(c, Rd)$, by

$$\varsigma^{M_{\mathbf{t}}}(f) := \varsigma^M(f)$$

where we use that we also have $f \in \mathcal{C}_n^T((c, \mathbf{t}), (Rd, \mathbf{t}))$ according to [Def. 2.26](#). Going the other way round, given a family $(M_{\mathbf{t}})_{\mathbf{t} \in T^n}$, we define the R_n -module M by

$$M(c, \mathbf{t}) := M_{\mathbf{t}}(c) .$$

Given a morphism $f \in \mathcal{C}_n^T((c, \mathbf{t}), (Rd, \mathbf{t}))$ — recall that morphisms in \mathcal{C}_n^T are only between families with *the same marker* \mathbf{t} —, we also have $f \in \mathcal{C}^T(c, Rd)$ and define

$$\varsigma^M(f) := \varsigma^{M_{\mathbf{t}}}(f) .$$

The remark extends to morphisms of modules; indeed, a morphism of modules $\alpha : M \rightarrow N$ on categories with pointed index sets corresponds to a family of morphisms $(\alpha_{\mathbf{t}} : M_{\mathbf{t}} \rightarrow N_{\mathbf{t}})_{\mathbf{t} \in T^n}$ between the associated families of modules.

As in [Sect. 3.2](#), we restrict our attention to half-arities which correspond, in a sense made precise below, to the syntactically defined arities of [Def. 3.31](#). The basic brick is the *tautological module of degree n* :

3.38 Definition: Given a category \mathcal{C} and $n \in \mathbb{N}$, any monad R on the category \mathcal{C}^T induces a monad R_n on \mathcal{C}_n^T with object map $(V, t_1, \dots, t_n) \mapsto (RV, t_1, \dots, t_n)$, as is already indicated for functors in [Def. 2.26](#).

3.39 Definition (Tautological Module of Degree n): Let $n \in \mathbb{N}$ be a natural number. To any S -monad R we associate the tautological module of R_n ,

$$\Theta_n(R) := (R_n, R_n) \in \text{LMod}_n(S, \text{Set}_n^T) .$$

This construction extends to a functor $\Theta_n : S\text{-Mnd} \rightarrow \text{LMod}_n(S, \text{Set}_n^T)$.

Let us consider the signature S_{TLC} of types of TLC. In the syntactically defined arities (cf. [Disp. \(3.3.2\)](#)) we write terms like $1 \rightsquigarrow 2$. We now give meaning to such a term: let T be any representation of S_{TLC} , that is, a set T together with a base type $* \in T$ and a binary operation $(\rightsquigarrow) : T \times T \rightarrow T$. Intuitively, the term $1 \rightsquigarrow 2$ should associate,

to an object (T, V, t_1, t_2) with a T -indexed family V of sets and $t_1, t_2 \in T$, the element $t_1 \rightsquigarrow t_2 \in T$. More formally, such a term is interpreted by a natural transformation (cf. Def. 3.41) over a specific category, whose objects are triples of a representation T of S_{TLIC} , a family of sets indexed by (the set) T and “markers” $(t_1, t_2) \in T^2$.

We go back to considering an arbitrary signature S for types. The following are the corresponding basic categories of interest:

3.40 Definition ($S\mathcal{C}_n$): Given a category \mathcal{C} — think of it as the category **Set** of sets — we define the category $S\mathcal{C}_n$ to be the category an object of which is a triple (T, V, \mathbf{t}) where T is a representation of S , the object $V \in \mathcal{C}^T$ is a T -indexed family of objects of \mathcal{C} and \mathbf{t} is a vector of elements of T of length n . We denote by $SU_n : S\mathcal{C}_n \rightarrow \mathbf{Set}$ the functor mapping an object (T, V, \mathbf{t}) to the underlying set T .

We have a forgetful functor $S\mathcal{C}_n \rightarrow \mathcal{T}\mathcal{C}_n$ which forgets the representation structure. On the other hand, any representation T of S in a set T gives rise to a functor $\mathcal{C}_n^T \rightarrow S\mathcal{C}_n$, which “attaches” the representation structure.

The meaning of a term $s \in S(n)$ as a natural transformation

$$s : 1 \Rightarrow SU_n : S\mathcal{C}_n \rightarrow \mathbf{Set}$$

is now given by recursion on the structure of s :

3.41 Definition (Canonical Natural Transformation): Let $s \in S(n)$ be a type of degree n . Then s denotes a natural transformation

$$s : 1 \Rightarrow SU_n : S\mathcal{C}_n \rightarrow \mathbf{Set}$$

defined recursively on the structure of s as follows: for $s = \alpha(a_1, \dots, a_k)$ the image of a constructor $\alpha \in S$ we set

$$s(T, V, \mathbf{t}) = \alpha(a_1(T, V, \mathbf{t}), \dots, a_k(T, V, \mathbf{t}))$$

and for $s = m$ with $1 \leq m \leq n$ we define

$$s(T, V, \mathbf{t}) = \mathbf{t}(m) .$$

We call a natural transformation of the form $s \in S(n)$ *canonical*.

Canonical natural transformations are used to build *classic* half-aritys; they indicate context extension (derivation) and selection of specific object types (fibre):

3.42 Definition (Classic Half-Arity over S): The following clauses define an inductive set of *classic* half-aritys, to which we restrict our attention:

- The constant functor $* : R \mapsto 1$ is a classic half-arity.

3. Simple Type Systems

- Given any canonical natural transformation $\tau : 1 \rightarrow SU_n$ (cf. Def. 3.41), the point-wise fibre module with respect to τ (cf. Def. 2.59) of the tautological module $\Theta_n : R \mapsto (R_n, R_n)$ (cf. Def. 3.39) is a classic half-arity of degree n ,

$$[\Theta_n]_\tau : S\text{-Mnd} \rightarrow \text{LMod}_n(S, \text{Set}) , \quad R \mapsto (R, [R_n]_\tau) .$$

- Given any (classic) half-arity $M = (M_1, M_2) : S\text{-Mnd} \rightarrow \text{LMod}_n(S, \text{Set})$ of degree n and a canonical natural transformation $\tau : 1 \rightarrow SU_n$, the point-wise derivation of M with respect to τ (cf. Def. 2.57) is a (classic) half-arity of degree n ,

$$M^\tau : S\text{-Mnd} \rightarrow \text{LMod}_n(S, \text{Set}) , \quad R \mapsto (M(R))^\tau := (M_1(R), M_2(R)^\tau) .$$

Here $(M(R))^\tau$ really means derivation of the module, i.e. derivation in the second component of $M(R)$.

- Given two (classic) half-arithies $M = (M_1, M_2)$ and $N = (N_1, N_2)$ of degree n , which coincide pointwise on the first component, i.e. such that $M_1 = N_1$. Then their product $M \times N$ is again a (classic) half-arity of degree n . Here the product is really the pointwise product in the second component, i.e.

$$M \times N : R \mapsto (M_1(R), M_2(R) \times N_2(R)) .$$

3.43 Remark *Classic Half-Arity, Syntactically:* We can represent a classic half-arity of degree $n \in \mathbb{N}$ over a signature S for types in a purely syntactic manner: such a half-arity is determined by a list of the form

$$[(\mathbf{t}_1, s_1), \dots, (\mathbf{t}_k, s_k)] ,$$

where \mathbf{t}_i are vectors of finite length of elements of $S(n)$ and $s_i \in S(n)$. Such a list corresponds precisely to the classic half-arity

$$R \mapsto [R_n]_{s_1}^{\mathbf{t}_1} \times \dots \times [R_n]_{s_k}^{\mathbf{t}_k} .$$

We use *weighted sets* as indexing sets for families of arities. The weight denotes the degree of the corresponding arity.

3.44 Definition (Weighted Set): A weighted set is a set J together with a map $d : J \rightarrow \mathbb{N}$.

An arity of degree $n \in \mathbb{N}$ for terms over an algebraic signature S is a pair of functors from S -monads to modules in $\text{LMod}_n(S, \text{Set})$. The degree n corresponds to the number of indices of its associated constructor. As an example, the arities of Abs and App of Ex. 1.3 are of degree 2, cf. Ex. 3.47.

3.45 Definition (Term–Arity, Signature over S): A *classic arity* α over S of degree n is a pair

$$s = (\text{dom}(\alpha), \text{cod}(\alpha))$$

of half-arithies over S of degree n such that

- $\text{dom}(\alpha)$ is classic and
- $\text{cod}(\alpha)$ is of the form $[\Theta_n]_\tau$ for some natural transformation τ as in Def. 3.42.

We write $\text{dom}(\alpha) \rightarrow \text{cod}(\alpha)$ for the arity α , and

$$\text{dom}(\alpha, R) := \text{dom}(\alpha)(R)$$

(and similarly for the codomain functor cod). Any classic arity is thus of the form given in Disp. (3.3.3). Given a weighted set (J, d) , a term–signature Σ over S indexed by (J, d) is a J -family Σ of algebraic arities over S , the arity $\Sigma(j)$ being of degree $d(j)$ for any $j \in J$.

Finally, a *typed signature* is a pair of a signature for types and a signature for terms over those types:

3.46 Definition (Typed Signature): A *typed signature* is a pair (S, Σ) consisting of an algebraic signature S and a term–signature Σ (indexed by some weighted set) over S .

3.47 Example (TLC, Ex. 1.3 continued): The terms of the simply typed lambda calculus over the type signature of Ex. 3.2 are given by the arities

$$\begin{aligned} \text{abs} &: [\Theta]_2^1 \rightarrow [\Theta]_{1 \rightsquigarrow 2} , \\ \text{app} &: [\Theta]_{1 \rightsquigarrow 2} \times [\Theta]_1 \rightarrow [\Theta]_2 , \end{aligned}$$

both of which are of degree 2 — we use the convention of Rem. 3.33. The outer lower index and the exponent are to be interpreted as de Bruijn variables, ranging over types. They indicate the fibre (cf. Def. 2.59) and derivation (cf. Def. 2.57), respectively, in the special case where the corresponding natural transformation is given by a natural number as in Def. 3.41. In particular, contrast that to the signature for the simply–typed lambda calculus we gave in Sect. 3.2, Ex. 3.23. The difference is that now “similar” arities which differ only in an object type parameter, are grouped together, whereas this is not the case in Ex. 3.23.

Those two arities can in fact be considered over any algebraic signature S with an arrow constructor, in particular over the signature S_{PCF} (cf. Ex. 3.48).

3.48 Example (Ex. 3.8 continued): We continue considering PCF. The signature S_{PCF} for its types is given in Ex. 3.4. The term–signature of PCF is given in Fig. 3.1: it consists of an arity for abstraction and an arity for application, each of degree 2, an arity (of degree 1) for the fixed point operator, and one arity of degree 0 for each logic and arithmetic constant — some of which we omit:

$$\begin{aligned}
&\text{abs} : [\Theta]_2^1 \rightarrow [\Theta]_{1 \Rightarrow 2} , \\
&\text{app} : [\Theta]_{1 \Rightarrow 2} \times [\Theta]_1 \rightarrow [\Theta]_2 , \\
&\text{Fix} : [\Theta]_{1 \Rightarrow 1} \rightarrow [\Theta]_1 , \\
&\mathbf{n} : * \rightarrow [\Theta]_\iota \quad \text{for } n \in \mathbb{N} \\
&\text{Succ} : * \rightarrow [\Theta]_{\iota \Rightarrow \iota} \\
&\text{Pred} : * \rightarrow [\Theta]_{\iota \Rightarrow \iota} \\
&\text{Zero?} : * \rightarrow [\Theta]_{\iota \Rightarrow o} \\
&\text{cond}_\iota : * \rightarrow [\Theta]_{o \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota} \\
&\mathbf{T}, \mathbf{F} : * \rightarrow [\Theta]_o \\
&\vdots
\end{aligned}$$

Figure 3.1.: Term Signature of PCF

Our presentation of PCF is inspired by Hyland and Ong’s [HO00], who — similarly to Plotkin [Plot77] — consider, e.g., the successor as a constant of arrow type. As an alternative, one might consider the successor as a constructor expecting a term of type ι as argument, yielding a term of type ι . For our purpose, those two points of view are equivalent.

3.3.2. Representations of Typed Signatures

A representation of a typed signature (S, Σ) is given by a representation of S (in a set) and a representation of Σ in a suitable monad:

3.49 Definition (Representation of a Signature over S): Let (S, Σ) be a typed signature. A *representation* R of (S, Σ) is given by

- an S -monad P and
- for each arity α of Σ , a morphism (in the large category of modules)

$$\alpha^R : \text{dom}(\alpha, P) \rightarrow \text{cod}(\alpha, P) ,$$

such that $\pi_1(\alpha^R) = \text{id}_P$.

In the following we also write R for the S -monad underlying the representation R . Note that the representation of S is “hidden” in the S -monad P .

A *morphism of representations* accordingly consists of a morphism of representations of S together with a morphism of representations of Σ , that is, a monad morphism that is compatible with the term representations:

3.50 Definition (Morphism of Representations): Given representations P and R of a typed signature (S, Σ) , a morphism of representations $f : P \rightarrow R$ is given by a morphism of S -monads $f : P \rightarrow R$, such that, for any arity α of Σ , the following diagram of module morphisms commutes:

$$\begin{array}{ccc} \text{dom}(\alpha, P) & \xrightarrow{\alpha^P} & \text{cod}(\alpha, P) \\ \text{dom}(\alpha, f) \downarrow & & \downarrow \text{cod}(\alpha, f) \\ \text{dom}(\alpha, R) & \xrightarrow{\alpha^R} & \text{cod}(\alpha, R). \end{array}$$

Again the morphism of representations of S is “hidden” in the morphism of S -monads.

3.51 Remark: Taking a 2-categoric perspective, the above diagram can be read as an equality of 2-cells

$$\begin{array}{c} \text{dom}(\alpha, P) \\ \downarrow \alpha^P \\ P \xrightarrow{\text{cod}(\alpha, P)} \text{Id}_{\text{Set}} \\ \downarrow cf \\ f^* \text{cod}(\alpha, R) \end{array} = \begin{array}{c} \text{dom}(\alpha, P) \\ \downarrow df \\ P \xrightarrow{f^* \text{dom}(\alpha, R)} \text{Id}_{\text{Set}} \\ \downarrow f^* \alpha^R \\ f^* \text{cod}(\alpha, R) \end{array},$$

where we write df and cf instead of $\text{dom}(\alpha, f)$ and $\text{cod}(\alpha, f)$, respectively.

The diagram of Def. 3.50 lives in the category $\text{LMod}_n(S, \text{Set})$ — where n is the degree of α — where objects are pairs (P, M) of a S -monad P of $S\text{-Mnd}_n$ and a module M over P . The above 2-cells are morphisms in the category $\text{Mod}(P_n, \text{Set})$, obtained by taking the second projection of the diagram of Def. 3.50. Note that for easier reading, we leave out the projection function and thus write $\text{dom}(\alpha, R)$ for the R_n -module of $\text{dom}(\alpha, R)$, i.e. for its second component, and similar elsewhere.

Representations of (S, Σ) and their morphisms form a category.

3.52 Remark: We obtain Zsidó's category of representations [Zsi10, Chap. 6] by restricting ourselves to representations of (S, Σ) whose type representation is the initial one. More, precisely, a signature (S, Σ) maps to a signature, say, $Z(S, \Sigma)$ over the initial set of sorts \hat{S} in the sense of Zsidó (cf. Sect. 3.2 and [Zsi10, Chap. 6]), obtained by

unbundling each arity of higher degree into a family of arities of degree 0. For instance, the signature of [Ex. 3.47](#) maps to the signature given in [Ex. 3.23](#). Representations of this latter signature in the sense of [Sect. 3.2](#) then are in one-to-one correspondence to representations in the sense of this section of the signature of [Ex. 3.47](#) over the initial representation \hat{S} of sorts, via the equivalence explained in [Rem. 3.37](#).

3.3.3. Initiality

We have all the ingredients to state and prove an initiality theorem for typed signatures:

3.53 Theorem: *For any typed signature (S, Σ) , the category of representations of (S, Σ) has an initial object.*

Proof. The proof consists of the following steps:

1. find the initial representation \hat{S} of the type signature S ;
2. define the monad $\hat{\Sigma}$ of terms specified by Σ on the category $\text{Set}^{\hat{S}}$;
3. equip the S -monad $\hat{\Sigma}$ with a representation structure of Σ , yielding a representation $\hat{\Sigma}$ of (S, Σ) ;
4. for any representation R of (S, Σ) , give a morphism of representations $i_R : \hat{\Sigma} \rightarrow R$;
5. prove uniqueness of i_R .

We go through these points:

1. We have already established (cf. [Lem. 3.7](#)) that there is an initial representation of sorts, which we call \hat{S} . Its underlying set is called \hat{S} as well.
2. The term monad we associate to (S, Σ) is the same as Zsidó's [[Zsi10](#), Chap. 6] in the sense of [Rem. 3.52](#), i.e. it is the term monad associated to $Z(S, \Sigma)$. The construction of this monad in a set-theoretic setting is described in Zsidó's thesis. We will give its definition in a type-theoretic setting.

In the following the natural transformations τ_i are in fact vectors of multiple transformations like those in [Rem. 2.30](#) (see also [Def. 2.57](#)), iterated by successive composition. Furthermore we make use of the simplified notation as introduced in [Not. 2.31](#).

We construct the monad which underlies the initial representation of (S, Σ) ,

$$\hat{\Sigma} : \text{Set}^{\hat{S}} \rightarrow \text{Set}^{\hat{S}} .$$

It associates to any set family of variables $V \in \text{Set}^{\hat{S}}$ an inductive set of terms with the following constructors:

- for every classic arity (of degree n)

$$\alpha = [\Theta_n]_{\sigma_1}^{\tau_1} \times \dots \times [\Theta_n]_{\sigma_m}^{\tau_m} \rightarrow [\Theta_n]_{\sigma} \quad (3.3.5)$$

we have a family of constructors indexed n times by $\mathbf{t} = (t_1, \dots, t_n)$ as well as by the context $V \in \text{Set}^{\hat{S}}$:

$$\alpha_{\mathbf{t}}(V) : \hat{\Sigma}^{\tau_1(V, \mathbf{t})}(V)_{\sigma_1(V, \mathbf{t})} \times \dots \times \hat{\Sigma}^{\tau_m(V, \mathbf{t})}(V)_{\sigma_m(V, \mathbf{t})} \rightarrow \hat{\Sigma}(V)_{\sigma(V, \mathbf{t})}$$

- a family of constructors

$$\text{Var}(V)_t : V_t \rightarrow \hat{\Sigma}(V)_t$$

indexed by contexts and the set \hat{S} of sorts.

The monadic structure is, accordingly, defined in the same way as in [Zsi10], by variables-as-terms — using the constructor Var — and flattening.

3. The representation structure on the monad $\hat{\Sigma}$ is defined by currying, and corresponds to Zsidó's: given an arity α of degree n in Σ , we must specify a module morphism

$$\alpha^{\hat{\Sigma}} : \text{dom}(\alpha, \hat{\Sigma}) \rightarrow \text{cod}(\alpha, \hat{\Sigma}) ,$$

where $\text{dom}(\alpha, \hat{\Sigma})$ and $\text{cod}(\alpha, \hat{\Sigma})$ are modules in $\text{Mod}(\hat{\Sigma}_n, \text{Set})$. We define

$$\alpha^{\hat{\Sigma}}(V, \mathbf{t})(a) := \alpha_{\mathbf{t}}(V)(a) ,$$

that is, the image under the constructor α from the definition of the monad $\hat{\Sigma}$. This yields a morphism of modules α of degree n ; note that according to [Rem. 3.37](#) it would be equivalent to specify a family $\alpha_{\mathbf{t}}^{\hat{\Sigma}}$ of module morphisms of suitable type, indexed by \mathbf{t} , which is actually done by Zsidó.

4. Given any other representation R over a set of sorts T , initiality of \hat{S} gives a “translation of sorts” $g : \hat{S} \rightarrow T$.

The morphism $i : \hat{\Sigma} \rightarrow R$ on terms is defined by structural recursion. Unfolding the definition of colax monad morphism, we need to define, for any context $V \in \text{Set}^{\hat{S}}$, a map of type

$$i_V : \forall t' \in T, \vec{g}(\hat{\Sigma}(V))_{t'} \rightarrow R(\vec{g}V)_{t'} .$$

Via the adjunction of [Def. 2.22](#) we equivalently define a map i as a family

$$i_V : \forall t \in \hat{S}, \hat{\Sigma}(V)_t \rightarrow R(\vec{g}V)_{g(t)} .$$

Let $a \in \hat{\Sigma}(V)_t$ be a term. In case $a = \text{Var}(V)_t(v)$ is the image of a variable $v \in V_t$, we map it to

$$i_V(\text{Var}(V)_t(v)) := \eta^R(\vec{g}V)(g(t))(\text{ctype}(v)) .$$

3. Simple Type Systems

Otherwise the term $a = \alpha_{\mathbf{t}}(V)(a_1, \dots, a_k) \in \hat{\Sigma}(V)_{\sigma(V, \mathbf{t})}$ is mapped to

$$i_V(\alpha_{\mathbf{t}}(V)(a_1, \dots, a_k)) := \alpha^R(\vec{g}(n)(V, \mathbf{t}))(i(a_1), \dots, i(a_k)) . \quad (3.3.6)$$

This map is well-typed: note that $\vec{g}(n)(V, \mathbf{t}) = (\vec{g}V, g_*(\mathbf{t}))$ by definition (Def. 2.28) and $\vec{g}(n)((V, \mathbf{t})^\tau) = (\vec{g}V, g_*(\mathbf{t}))^\tau$, i.e. context extension and retyping permute.

The axioms of monad morphisms, i.e. compatibility of this map with respect to variables-as-terms and flattening are easily checked: the former is a direct consequence of the definition of i on variables, and the latter is proved by structural induction. This definition yields a morphism *of representations*; consider the arity α of Σ . For this arity, the commutative diagram of Def. 3.50 informally reads as follows: one starts in the upper-left corner with a tuple of terms, say, (a_1, \dots, a_k) of $\hat{\Sigma}$. Taking the upper-right path corresponds to the translation of the image of this tuple under the map $\alpha^{\hat{\Sigma}}$, i.e. under the constructor α of $\hat{\Sigma}$. The lower-left path corresponds to the image under the module morphism α^R of the translated tuple $(i(a_1), \dots, i(a_k))$. The diagram thus precisely states the equality of Disp. (3.3.6). We thus establish that i is (the carrier of) a morphism of representations $(g, i) : (\hat{S}, \hat{\Sigma}) \rightarrow R$.

5. Uniqueness of the morphism $i : (\hat{S}, \hat{\Sigma}) \rightarrow R$ is proved making use of the commutative diagram of Def. 3.50. Suppose that $(g', i') : (\hat{S}, \hat{\Sigma}) \rightarrow R$ is a morphism of representations. We already know that $g = g'$ by initiality of \hat{S} .

By structural induction on the terms of $\hat{\Sigma}$ we prove that $i = i'$: using the same notation as above, for $a = \alpha_{\mathbf{t}}(V)(a_1, \dots, a_k)$ we have

$$i'(a) = \alpha^R(i'(a_1), \dots, i'(a_k)) \stackrel{i(a_i) = i'(a_i)}{=} \alpha^R(i(a_1), \dots, i(a_k)) = i(a) .$$

In case $a = \text{Var}(v)$ is a variable, considered as a term, the fact that both i and i' are monad morphisms ensures that $i(\text{Var}(v)) = i'(\text{Var}(v)) = \eta_{\vec{g}V}^R(\text{ctype}(v))$. Thus we have proved $i = i'$.

□

The proof shows that the initial morphism to a representation R depends on the representation structure on R and not just on the monad R itself. We illustrate this on the example of the typed signature of PCF:

3.54 Example: Representing the signature of PCF in the untyped lambda calculus leaves one with several choices to take, e.g., as to how to translate the fixed point operator **Fix**. To represent **Fix** in ULC, one must give a unary operation on ULC. Reasonable from the semantic viewpoint are, e.g., the representations

$$x \mapsto \text{App}(\mathbf{Y}, x) \quad \text{or} \quad x \mapsto \text{App}(\Theta, x) , \quad (3.3.7)$$

using, e.g., one of the fixedpoint combinators

$$\begin{aligned} \mathbf{Y} &:= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \quad (\text{Curry}) \\ \Theta &:= (\lambda x.\lambda y.(y(xxy)))(\lambda x.\lambda y.(y(xxy))) \quad (\text{Turing}). \end{aligned}$$

By initiality, those two representations yield two different compilations of PCF to ULC, mapping a PCF term of the form $\mathbf{Fix}(f)$ to $\mathbf{Y}(f) = \text{App}(\mathbf{Y}, f)$ and $\Theta(f) = \text{App}(\Theta, f)$, respectively. The representation module morphisms thus constitute the “extra structure” ϕ , ψ and ψ' mentioned in Sect. 1.1. A complete translation is given in Chapt. 9.

3.4. Logics and Logic Translations

In the style of the Curry–Howard isomorphism, we consider propositions as types and proofs of a proposition as terms of that type. In this example we present the typed signatures of two different logics,

- Classical propositional logic, called **CPC**, and
- Intuitionistic propositional logic, called **IPC**.

According to our main theorem each of those signatures gives rise to an initial representation, a logical type system. We then use the *iteration principle* on **CPC** in order to specify a translation of *propositions and their proofs* from **CPC** to **IPC**. The translation we specify is actually the propositional fragment of the *Gödel–Gentzen negative translation* [TvD88, Def. 3.4].

3.4.1. Signatures of Classical and Intuitionistic Logic

We present typed signatures for classical and intuitionistic propositional logic. Their respective signatures for types — *propositions* — are the same: let P denote a set of atomic formulas. The *types* — propositions — of classical (**CPC**) and intuitionistic (**IPC**) propositional logic are given by the following algebraic signature:

$$\mathcal{P} := \{p : 0, \quad \top : 0, \quad \wedge : 2, \quad \perp : 0, \quad \vee : 2, \quad \Rightarrow : 2\}.$$

where for any atomic formula $p \in P$ we have an arity $p : 0$. We call \mathcal{P} the initial representation as well as its underlying set, i.e. the propositions of **CPC** and **IPC**. For the set \mathcal{P} we use infix binary constructors. Note that negation is defined as $\neg A \equiv A \Rightarrow \perp$.

3.4.1.1. Signature of CPC

For the *terms* of **CPC**, each inference rule is given by an arity. In Fig. 3.2 (p. 82), the inference rules and their corresponding arities are presented. Each inference rule

3. Simple Type Systems

corresponds to a (family of) term — *proof* — constructor(s), where inference rules without hypotheses are constants. Note that the initial representation automatically comes with an additional inference rule

$$\frac{}{\Gamma, A \vdash A} \text{var}$$

corresponding to the monadic operation η , i.e. to the variables-as-terms constructor. Analogously to [Rem. 3.24](#), it is not necessary, using our approach, to specify this inference rule explicitly by an arity in the term signature of the logic under consideration; any logic we specify via a typed signature automatically comes with this rule.

3.4.1.2. Signature of IPC

The type signature and thus the formulas of intuitionistic propositional logic **IPC** are the same as for **CPC**. However, the *term* signature is missing the arity EM for excluded middle.

3.4.2. Translation via Initiality

The translation of propositions $(_)^g : \mathcal{P} \rightarrow \hat{\mathcal{P}}$, i.e. on the *type* level, is specified by a representation g of the algebraic signature \mathcal{P} in the set $\hat{\mathcal{P}}$. According to [Def. 3.3](#) we must specify, for any arity $s : n \in \mathbb{N}$ of \mathcal{P} , a map towards $\hat{\mathcal{P}}$ taking a suitable number of arguments in $\hat{\mathcal{P}}$,

$$s^g : \hat{\mathcal{P}}^n \rightarrow \hat{\mathcal{P}}.$$

There is, of course, a canonical such map for each arity — but this would only give us the identity morphism on $\hat{\mathcal{P}}$. We represent \mathcal{P} in $\hat{\mathcal{P}}$ not by this identity representation, but in such a way that we obtain the Gödel–Gentzen negative translation:

$$\begin{aligned} p^g &:= \neg\neg p, & \top^g &:= \neg\neg\top, & \wedge^g &:= \wedge, & \vee^g &:= (A, B) \mapsto \neg(\neg A \wedge \neg B), \\ \Rightarrow^g &:= (\Rightarrow), & \perp^g &:= \neg\neg\perp. \end{aligned}$$

The proofs of **IPC** are given by the signature of **CPC** without the classical axiom EM. We represent EM in **IPC** by giving, for any proposition A , a term of type $\neg(\neg\neg A \wedge \neg A)$, e.g.,

$$\frac{\frac{\frac{}{\neg\neg A \wedge \neg A \vdash \neg\neg A \wedge \neg A} \text{var}}{\neg\neg A \wedge \neg A \vdash \neg\neg A} \wedge_{E1}}{\neg\neg A \wedge \neg A \vdash \perp} \Rightarrow_E \quad \frac{}{\vdash \neg\neg A \wedge \neg A \Rightarrow \perp} \Rightarrow_I$$

As another example, we give a representation of \vee_{II} , that is, for any proposition A and B , we give a term of type $A^g \rightarrow \neg(\neg A^g \wedge \neg B^g)$:

$$\frac{\frac{\frac{A^g}{\neg\neg A^g}}{\neg\neg A^g \vee \neg\neg B^g} \vee_{I1}}{\neg(\neg A^g \wedge \neg B^g)} \text{De Morgan}$$

Here the proof of $A^g \rightarrow \neg\neg A^g$ and of the used De Morgan law are abbreviations for longer proofs in **IPC**. We leave it up to the reader to find representations in **IPC** for the other arities.

3.4.3. Remarks

This representation of the signature of **CPC** in **IPC** yields the (propositional fragment of the) Gödel–Gentzen translation of propositions specified in Troelstra and van Dalen’s book [TvD88, Def. 3.4], denoted on propositions with the same name as its specifying representation,

$$(_)^g : \mathcal{P} \rightarrow \mathcal{P} .$$

Our translation of terms shows that any provable proposition in **CPC** translates to a provable proposition in **IPC**, since we provide the corresponding proof term via our translation:

$$\Gamma \vdash_{\mathbf{C}} A \text{ implies } \Gamma^g \vdash_{\mathbf{I}} A^g .$$

However, a logic translation t from a logic **L** to another logic **L'** should certainly satisfy an *equivalence* of the form

$$\Gamma \vdash_{\mathbf{L}} A \text{ if and only if } \Gamma^t \vdash_{\mathbf{L}'} A^t .$$

Our framework does *not* ensure the implication from right to left, and is thus deficient from the point of view of logic translations.

Another important property of logics is normalization through cut elimination. This aspect can be treated using the techniques presented in [Chapt. 5](#), where we integrate *reduction rules* into the notion of signature and their representations as presented in this chapter.

3. Simple Type Systems

| Inference Rule | Arity |
|---|---|
| $\frac{}{\Gamma \vdash \top} \top_I$ | $\top_I : * \rightarrow [\Theta]_{\top}$ |
| $\frac{\Gamma \vdash \perp}{\Gamma \vdash A} \perp_I$ | $\perp_I : [\Theta]_{\perp} \rightarrow [\Theta]_1$ |
| $\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_I$ | $\wedge_I : [\Theta]_1 \times [\Theta]_2 \rightarrow [\Theta]_{1 \wedge 2}$ |
| $\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_{E1}$ | $\wedge_{E1} : [\Theta]_{1 \wedge 2} \rightarrow [\Theta]_1$ |
| $\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_{E2}$ | $\wedge_{E1} : [\Theta]_{1 \wedge 2} \rightarrow [\Theta]_2$ |
| $\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow_I$ | $\Rightarrow_I : [\Theta]_2^1 \rightarrow [\Theta]_{1 \Rightarrow 2}$ |
| $\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow_E$ | $\Rightarrow_E : [\Theta]_{1 \Rightarrow 2} \times [\Theta]_1 \rightarrow [\Theta]_2$ |
| $\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_{I1}$ | $\vee_{I1} : [\Theta]_1 \rightarrow [\Theta]_{1 \vee 2}$ |
| $\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_{I2}$ | $\vee_{I2} : [\Theta]_2 \rightarrow [\Theta]_{1 \vee 2}$ |
| $\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_E$ | $\vee_E : [\Theta]_{1 \vee 2} \times [\Theta]_3^1 \times [\Theta]_3^2 \rightarrow [\Theta]_3$ |
| $\frac{}{\Gamma \vdash \neg A \vee A} \text{EM}$ | $\text{EM} : * \rightarrow [\Theta]_{\neg 1 \vee 1}$ |

Figure 3.2.: Inference Rules of **CPC** and their Arities

4. REDUCTIONS FOR UNTYPED SYNTAX

We now would like to consider not just the *terms* (and types) of a language, but also *reductions* on the terms. As an example, suppose we would like to equip the untyped lambda calculus with the reduction relation generated by the beta rule given in [Disp. \(A.2.1\)](#). We could produce the syntax associated to the signature via the universal property explained in the preceding section — possibly in a computer implementation thereof — and define a suitable relation on the terms of the language *a posteriori*.

However, in this way we would not have any guarantee concerning compatibility of substitution with respect to this reduction relation. Furthermore, how could we ensure any compatibility of a translation from the initial representation to another term language, equipped with some reduction rules, specified via the iteration principle? There would not be any systematic way of doing so, we would need to check manually for each translation we consider.

The solution to this problem is to integrate reduction rules into signatures and the models of those signatures. Indeed, instead of considering reduction rules for just the initial representation of a signature, say, Σ , we define *inequations over Σ* , which specify rules for *each* representation of Σ . However, not all of the representations of Σ satisfy those rules; we define a “satisfaction” predicate on the representations of Σ , to pick out the representations that satisfy those rules.

In order to define the satisfaction predicate, we need to consider representations whose codomain (read: the codomain of the underlying monad) is not the category of plain sets, but of sets with a structure suitable to express relations between its elements. The following monadic models come to mind:

- X $M : \text{Set} \rightarrow \text{Set}$ — Terms modulo relations by quotienting
We reject the idea of quotienting by the congruence relation generated by a set of inequations on the grounds that we want to avoid adding a *symmetry* rule and thus loose the information of *direction* of a reduction
- X $M : \text{Pre} \rightarrow \text{Pre}$ — Monads on preordered sets
While the use of monads on preordered sets allows to retain directions of reductions, it would necessitate to consider *preordered* contexts. However, contexts usually are given by *unstructured* sets of variables.
- ✓ $M : \text{Set} \rightarrow \text{Pre}$ — *Relative Monads* from sets to preordered sets
Relative monads from sets to preorders avoid the problems one encounters with

4. Reductions for Untyped Syntax

the aforementioned approaches. As shown in [Sect. 2.4.1](#), the mediating functor to use is the functor $\Delta : \text{Set} \rightarrow \text{Pre}$.

Before going into more detail concerning the models of signatures with inequations, we have a closer look at those signatures themselves. Signatures should carry information about

Syntax the terms, optionally typed over a set of sorts, and

Semantics *reductions* on the terms.

Accordingly, we introduce a notion of *2-signature*. A 2-signature (Σ, A) consists of a (higher-order) signature Σ — which we also call 1-signature from now on, to emphasize the existence of a second level, the *semantic* level — which specifies the terms of a language, as well as a set A of *inequations* over Σ . Each inequation of A specifies a reduction rule.

We borrow the terms “1-signature” and “2-signature” from T. Hirschowitz [[Hir](#)]: they are motivated by the point of view of Categorical Semantics. There, types and terms of a language are modelled as the objects and morphisms of a category. Furthermore, reductions between terms may be modelled through 2-cells. In this way, a 1-signature specifies a 1-category, whereas a 2-signature specifies a 2-category.

As the 1-signature which underlies a 2-signature, we may choose any of the notions of signature defined in the preceding chapters (cf. [Defs. 3.18, 3.46](#)). For this chapter, however, we restrict ourselves to *untyped* syntax with reductions, allowing us to employ a simple notion of 1-signature. The next chapter integrates reductions and types.

While we present 1-signatures from two perspectives, a syntactic one and a semantic one, we only present inequations semantically. We refer to [Sect. 10.2](#) for thoughts about the syntactic aspect.

4.1. 1-Signatures

We start out by defining *1-signatures* in two different ways, once syntactically, and once in terms of pairs of functors between suitable categories.

The syntactic description of arities is actually the same as in [Sect. 3.2](#), even simpler: since we only consider *untyped* syntax, we just need to specify the number of arguments of a constructor, and, for each argument, the number of variables bound in it:

4.1 Definition (Classic Arity, Signature): A *classic arity* is given by a list of natural numbers. The length of the list indicates the number of arguments of its associated constructor, whereas the i -th component of the list specifies the number of variables bound in the i -th argument. A *classic signature* is given by a family of arities.

4.2 Example (Untyped Lambda Calculus): The signature of the untyped lambda calculus is given by

$$\Sigma_{\text{ULC}} := \{\text{app} : [0,0] , \quad \text{abs} : [1]\} .$$

For the semantic definition of arities, we define a suitable category of monads and a large category of modules. As discussed at the beginning of the chapter, we use *relative* monads and modules over relative monads.

We start by giving a simplified version of the definition of morphism of relative monads, to which we restrict ourselves throughout this chapter. It is obtained from [Def. 2.87](#) by restricting the vertical functors G and G' to the identity functor. Furthermore we will have $F = F'$, and the natural transformation N is the identity transformation. Given two relative monads P and Q on $F : \mathcal{C} \rightarrow \mathcal{D}$, a (*simple*) *morphism of relative monads* is a family of morphisms $\tau_c \in \mathcal{D}(Pc, Qc)$ that is compatible with the monadic structure:

4.3 Definition (Morphism of Relative Monads): Given two relative monads P and Q from \mathcal{C} to \mathcal{D} on the functor $F : \mathcal{C} \rightarrow \mathcal{D}$, a *morphism of monads* from P to Q is given by a collection of morphisms $\tau_c \in \mathcal{D}(Pc, Qc)$ such that the following diagrams commute for all suitable morphisms f :

$$\begin{array}{ccc} Pc & \xrightarrow{\sigma^P(f)} & Pd \\ \tau_c \downarrow & & \downarrow \tau_d \\ Qc & \xrightarrow{\sigma^Q(\tau_d \circ f)} & Qd \end{array} \qquad \begin{array}{ccc} Fc & \xrightarrow{\eta_c^P} & Pc \\ & \searrow \eta_c^Q & \downarrow \tau_c \\ & & Qc. \end{array}$$

As a consequence from these commutativity properties the family τ is a natural transformation between the functors induced by the monads P and Q (cf. [Rem. 2.80](#)).

4.4 Definition (Category of Relative Monads on F): Given a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, we define the category $\text{RMon}(F)$ to be the category whose objects are relative monads on F . A morphism from P to Q in $\text{RMon}(F)$ is a morphism as in [Def. 4.3](#).

There is an adjunction between relative monads on Δ and monads on sets:

4.5 Lemma (Adjunction between $\text{Mon}(\text{Set})$ and $\text{RMon}(\Delta)$): *The functors (with object functions) defined in [Lem. 2.83](#) give rise to an adjunction*

$$\begin{array}{ccc} & \Delta_* & \\ \text{Mon}(\text{Set}) & \begin{array}{c} \xrightarrow{\quad} \\ \perp \\ \xleftarrow{\quad} \end{array} & \text{RMon}(\Delta) \\ & U_* & \end{array} .$$

4. Reductions for Untyped Syntax

Proof. The isomorphism $\varphi_{PQ} : \text{RMon}(\Delta)(\Delta_*P, Q) \cong \text{Mon}(\text{Set})(P, U_*Q)$ is defined by applying the adjunction of [Lem. 2.18](#) in each morphism of the family underlying a morphism of (relative) monads. Commuting diagrams are not modified by applying this adjunction. Naturality of φ is trivial. \square

4.6 Definition (Large Category of Modules): Given a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ and a category \mathcal{E} , we define the category $\text{LRMod}(F, \mathcal{E})$ to be the category whose objects are pairs (P, M) of a relative monad $P \in \text{RMon}(F)$ and a relative P -module M with codomain \mathcal{E} . A morphism to another such (Q, N) is a pair (h, f) of a morphism $h : P \rightarrow Q$ in $\text{RMon}(F)$ and a morphism of P -modules $f : P \rightarrow h^*Q$ to the pullback of Q along h (cf. [Sect. 2.4.2](#)).

For any monad P on F there is the injection functor

$$I_P : \text{RMod}(P, \mathcal{E}) \rightarrow \text{LRMod}(F, \mathcal{E}), \quad f \mapsto (\text{id}, f) .$$

A *half-arity* associates a P -module towards the category Pre of preorders to any relative monad P on Δ :

4.7 Definition (Half-Arity): A *half-arity* a is a functor

$$a : \text{RMon}(\Delta) \rightarrow \text{LRMod}(\Delta, \text{Pre})$$

that is pre-inverse to the forgetful functor.

Similarly to the preceding sections we restrict our attention to *classic* half-arithies:

4.8 Definition (Classic Half-Arity): The following clauses define the inductive set of *classic* half-arithies:

- $\Theta : P \mapsto (P, P)$, the tautological module, is classic;
- if M is classic, so is its derivation $M' : P \mapsto (P, M(P)')$;
- if M and N are classic, so is their product $M \times N : P \mapsto (P, M(P) \times N(P))$;
- the constant half-arity $*$: $P \mapsto 1$ is classic.

Classic half-arithies as defined in [Def. 4.8](#) are in one-to-one correspondence to classic arities as defined in [Def. 4.1](#):

4.9 Remark: We use the notation defined in [Not. 2.103](#). More generally, given a list of natural numbers $s = [n_1, \dots, n_m]$, we write $M^s := M^{n_1} \times M^{n_2} \times \dots \times M^{n_m}$.

The same notation is used for morphisms, i.e. given a morphism of R -modules $f : M \rightarrow N$, we write

$$f^s := f^{n_1} \times \dots \times f^{n_m} : M^s \rightarrow N^s .$$

Thus any list of natural numbers specifies uniquely a classic half-arity, the empty list denoting the terminal module $*$: $R \mapsto 1$.

4.10 Definition (Arity): An *arity* s is a pair $s = (\text{dom}(s), \text{cod}(s))$ of half-arities

$$\text{dom}(s), \text{cod}(s) : \text{RMon}(\Delta) \rightarrow \text{LRMod}(\Delta, \text{Set}) .$$

We write $s = \text{dom}(s) \rightarrow \text{cod}(s)$, and $\text{dom}(s, P) := \text{dom}(s)(P)$ (and similarly for cod).

4.11 Definition (Classic Arity, 1-Signature): A *classic arity* is an arity of the form

$$\text{dom}(s) \rightarrow \Theta$$

such that $\text{dom}(s)$ is a classic half-arity. Any classic arity as in [Def. 4.1](#) uniquely specifies a classic arity by specifying its domain according to [Rem. 4.9](#). A *1-signature* is a family of classic arities, or, equivalently according to [Rem. 4.9](#), a family of lists of natural numbers.

4.12 Example (Untyped Lambda Calculus): The 1-signature Σ_{ULC} of the untyped lambda calculus, already given syntactically in [Ex. 4.2](#), is given by the two arities

$$\text{app} := \Theta \times \Theta \rightarrow \Theta , \quad \text{abs} := \Theta' \rightarrow \Theta .$$

4.2. Representations of 1-Signatures

A representation of a classic arity s in a monad P is a module morphism $\text{dom}(s, P) \rightarrow P$. More generally:

4.13 Definition (Representation of an Arity): A representation of an arity $s = \text{dom}(s) \rightarrow \text{cod}(s)$ in a monad P on Δ is a morphism M of P -modules

$$M : \text{dom}(s, P) \rightarrow \text{cod}(s, P)$$

in the category $\text{LRMod}(\Delta, \text{Set})$, such that $\pi_1(M) = \text{id}$. By abuse of notation, we also denote by M the second projection of M , i.e. we consider $M \in \text{RMod}(P, \text{Set})$.

A representation of a signature is given by a relative monad on Δ and a representation of each arity in this monad:

4.14 Definition (Representation of a 1-Signature): A *representation* R of a signature Σ is given by

- a monad P on Δ and
- a representation $s^R : \text{dom}(s, P) \rightarrow \text{cod}(s, P)$ of each arity $s \in \Sigma$ in P as in [Def. 4.13](#).

Given a representation R , we denote its underlying monad by R as well.

4. Reductions for Untyped Syntax

For any signature Σ as in [Def. 4.11](#), we have representations of Σ in monads on \mathbf{Set} (cf. [Def. 3.25](#)) and in relative monads on Δ (cf. [Def. 4.14](#)). The following definition links those representations:

4.15 Definition (Reps. in Relative Monads and Monads): To any representation of a classic signature Σ in a relative monad R as defined in [Def. 4.14](#) we associate a representation of Σ in the monad U_*R (cf. [Lem. 4.5](#)) according to the definition of representation of [Def. 3.25](#), by postcomposing with the forgetful functor from preorders to sets.

Conversely, to any representation of Σ in a monad Q over sets we associate a representation of Σ in the relative monad Δ_*Q over Δ , by postcomposing with Δ . More precisely, an arity $s = [s_1, \dots, s_n] \in \Sigma$ and a representation of s in Q , say,

$$s^Q : Q^s \rightarrow Q ,$$

with $Q^s := Q^{s_1} \times \dots \times Q^{s_n}$, we have to give a morphism of modules

$$\Delta_*Q^{s_1} \times \dots \times \Delta_*Q^{s_n} \rightarrow \Delta_*Q ,$$

that is, a family of monotone morphisms in the category \mathbf{Pre} . However, the domain module is isomorphic to Δ_*Q^s , hence postcomposing the map s^Q with Δ does the job,

$$\Delta_*s^Q : \Delta_*Q^s \rightarrow \Delta_*Q ,$$

and Δ_*s^Q obviously has the necessary commutation property with respect to substitution.

4.16 Example ([Ex. 4.12](#) continued): A representation P of Σ_{ULC} is given by

- a monad $P : \mathbf{Set} \xrightarrow{\Delta} \mathbf{Pre}$ and
- two morphisms of P -modules in $\mathbf{RMod}(P, \mathbf{Pre})$,

$$\text{app} : P \times P \rightarrow P \quad \text{and} \quad \text{abs} : P' \rightarrow P .$$

Morphisms of representations are monad morphisms which commute with the representation morphisms of modules:

4.17 Definition (Morphism of Representations): Let P and Q be representations of a classic signature Σ . A *morphism of representations* $f : P \rightarrow Q$ is a morphism of monads $f : P \rightarrow Q$ such that the following diagram commutes for any arity $s \in \Sigma$:

$$\begin{array}{ccc} \text{dom}(s, P) & \xrightarrow{s^P} & P \\ \text{dom}(s, f) \downarrow & & \downarrow f \\ \text{dom}(s, Q) & \xrightarrow{s^Q} & Q. \end{array}$$

The meaning of those diagrams might become clearer when we consider the example of the untyped lambda calculus. In line with the abuse of notation mentioned in [Def. 4.13](#), we omit the first component of objects and morphisms in $\text{LRMod}(\Delta, \text{Set})$:

4.18 Example ([Ex. 4.16](#) continued): Let P and R be two representations of Σ_{ULC} . A morphism from P to R is given by a morphism of monads $f : P \rightarrow R$ such that the following diagrams of P -module morphisms commute:

$$\begin{array}{ccc}
 P \times P & \xrightarrow{\text{app}^P} & P \\
 f \times f \downarrow & & \downarrow f \\
 f^*(R \times R) & \xrightarrow{f^*(\text{app}^R)} & f^*R
 \end{array}
 \qquad
 \begin{array}{ccc}
 P' & \xrightarrow{\text{abs}^P} & P \\
 f' \downarrow & & \downarrow f \\
 f^*R' & \xrightarrow{f^*(\text{abs}^R)} & f^*R.
 \end{array}$$

To make sense of these diagram it is necessary to recall the constructions on modules of [Sect. 2.4.2](#). The diagrams live in the category $\text{RMod}(P, \text{Pre})$. The vertices are obtained from the tautological modules P resp. the Q over the monads P resp. Q by applying the pullback (for Q) and derivation functors as well as by the use of the product in the category of P -modules into Pre . The vertical morphisms are module morphisms induced by f , to which — on the left-hand side — functoriality of derivation and products are applied. Furthermore instances of [Lem. 2.106](#) and [2.107](#) are hidden in the lower left corner. The lower horizontal morphism makes use of the functoriality of the pullback operation.

4.19 Definition (Category of Representations): Representations of Σ and their morphisms form a category $\text{Rep}^\Delta(\Sigma)$.

4.20 Lemma (Adj. between Reps. in Rel. Monads and Reps. in Monads): *The assignment of [Def. 4.15](#) extends to an adjunction between the category of representations in relative monads on Δ and the category of representations in monads on sets (cf. [Def. 3.27](#)):*

$$\begin{array}{ccc}
 & \Delta_* & \\
 \text{Rep}(\Sigma) & \begin{array}{c} \curvearrowright \\ \perp \\ \curvearrowleft \end{array} & \text{Rep}^\Delta(\Sigma) \\
 & U_* &
 \end{array}
 .$$

4.21 Lemma (Initiality for 1-Signatures): *The category of representations of a signature Σ in relative monads as defined in [Def. 4.19](#) has an initial object. Its underlying monad associates, to any set of variables, the set of terms of Σ , equipped with the equality preorder.*

Proof. This is a direct consequence of [Lem. 2.19](#) which says that left adjoints preserve colimits — thus, in particular, initial objects —, applied to the adjunction of [Lem. 4.20](#). \square

4.3. Inequations

Consider the beta rule of lambda calculus,

$$\lambda M(N) \rightsquigarrow M[* := N] .$$

In our formalism, abstraction and application are considered as morphisms of modules (cf. Ex. 2.95), and so is substitution (cf. Def. 2.110). This suggests to define (in)equations over a 1-signature Σ as *parallel pairs* of module morphisms, indexed by representations of Σ . Put differently, an (in)equation associates a parallel pair of module morphisms to any representation of Σ . Hirschowitz and Maggesi [HM07b] specify equations through such pairs of (indexed) module morphisms over (plain) monads. We adapt their definition to our use of *relative* monads and modules over such monads. Afterwards we simply interpret a pair of half-equations as *inequation* rather than equation.

4.22 Definition (Category of Half-Equations, [HM07b]): Let Σ be a signature. A Σ -module U is a functor from the category of representations of Σ to the category $\text{LRMod}(\Delta, \text{wPre})$ commuting with the forgetful functors to the category of relative monads over Δ :

$$\begin{array}{ccc} \text{Rep}^\Delta(\Sigma) & \xrightarrow{U} & \text{LRMod}(\Delta, \text{wPre}) \\ & \searrow & \swarrow \\ & \text{RMon}(\Delta) & \end{array}$$

Such a Σ -module U associates, to any representation of Σ with underlying monad P , a module over P .

We define a morphism of Σ -modules to be a natural transformation which becomes the identity when composed with the forgetful functor. We call these morphisms *half-equations*. These definitions yield a category which we call the *category of Σ -modules* (or the *category of half-equations*). We sometimes write

$$U_X^R := U(R)(X)$$

for the value of a Σ -module at the representation R and the set X . Similarly, for a half-equation $\alpha : U \rightarrow V$ we write

$$\alpha_X^R := \alpha(R)(X) : U_X^R \rightarrow V_X^R .$$

4.23 Remark: We define Σ -modules over the signature Σ as functors into the category $\text{LRMod}(\Delta, \text{wPre})$, whose objects are modules *with codomain category wPre instead of Pre* to accommodate an important example: recall that substitution of *one* variable (cf. Def. 2.110) is not necessarily monotone in the second argument. Thus, in order to build a half-equation from this substitution (cf. Def. 4.27), we need to use the category wPre as codomain category.

4.24 Remark: A half-equation α from Σ -module U to V associates, to any representation R , a morphism of R -modules $\alpha^R : U(R) \rightarrow V(R)$ in $\mathbf{RMod}(R, \mathbf{wPre})$ such that for any morphism $f : P \rightarrow R$ of representations of Σ the following diagram commutes:

$$\begin{array}{ccc} (P, U(P)) & \xrightarrow{\alpha^P} & (P, V(P)) \\ \downarrow (f, U(f)) & & \downarrow (f, V(f)) \\ (R, f^*(U(R))) & \xrightarrow{\alpha^R} & (R, f^*(V(R))) \end{array} .$$

4.25 Remark: Pierre-Louis Curien suggested the following alternative definition of a half-equation, where its domain and codomain only depend on *the monad underlying each representation*: domain and codomain are specified by functors U and V on the category $\mathbf{RMon}(\Delta)$, and a half-equation α from U to V is given by a natural transformation

$$\alpha : U \circ \pi_1 \rightarrow V \circ \pi_1 ,$$

where $\pi_1 : \mathbf{Rep}^\Delta(\Sigma) \rightarrow \mathbf{RMon}(\Delta)$ is the forgetful functor. Indeed, in all the examples of half-equations we consider, the domain and codomain Σ -modules only depend on the monads underlying a representation, not the representation structure itself. Both variants, the one presented here in detail as well as the one suggested by Curien, are implemented in our Coq library.

Given a 1-signature Σ , we restrict ourselves to *classic* inequations: these are inequations whose codomain Σ -module is of a specific form. The restriction to these inequations allows us to ensure a technical condition which we prove, for classic inequations, in [Lem. 4.35](#). Analogously to the preceding chapters, we only write the second component of objects in the large category $\mathbf{LRMod}(\Delta, \mathbf{wPre})$ of modules.

4.26 Definition (Classic Σ -Module): We call *classic* any Σ -module satisfying the following inductive predicate.

- The map $\hat{\Theta} : R \mapsto \widehat{\pi_1 R}$ (cf. [Def. 2.109](#) and [Rem. 4.25](#)) is a classic Σ -module.
- If the Σ -module $M : R \mapsto M(R)$ is classic, so is

$$M' : R \mapsto M(R)' .$$

- If M and N are classic, so is

$$M \times N : R \mapsto M(R) \times N(R) .$$

- The terminal module $*$: $R \mapsto 1$ is classic.

4. Reductions for Untyped Syntax

Using the same notation as in [Rem. 4.9](#), any list of natural numbers specifies uniquely a classic Σ -module.

We now present some particular classic half-equations:

4.27 Definition: The substitution operation of [Def. 2.110](#),

$$\text{subst} : R \mapsto \text{subst}^R : \hat{R}' \times \hat{R} \rightarrow \hat{R}$$

is a half-equation over any 1-signature Σ . Its domain and codomain are classic.

4.28 Example ([Ex. 4.12](#) continued): The map

$$\text{app} \circ (\text{abs} \times \text{id}) : R \mapsto \text{app}^R \circ (\text{abs}^R \times \text{id}^R) : \hat{R}' \times \hat{R} \rightarrow \hat{R}$$

is a half-equation over the signature Σ_{ULC} .

4.29 Definition: Any arity $s = [n_1, \dots, n_m] \in \Sigma$ defines a classic Σ -module

$$\text{dom}(s) : R \mapsto R^{n_1} \times \dots \times R^{n_m} .$$

An *inequation* is given by a pair of parallel half-equations:

4.30 Definition (Inequations, 2-Signature): Given a 1-signature Σ , a Σ -*inequation* is a pair of parallel half-equations between Σ -modules. We write

$$\alpha \leq \gamma : U \rightarrow V$$

for the inequation (α, γ) with domain U and codomain V . A 2-signature is a pair (Σ, A) of a 1-signature Σ and a set A of Σ -inequations.

Given a 2-signature (Σ, A) , we can test whether a given representation R of Σ satisfies the inequations of A . Those representations satisfying any inequation of A form the category of representations of (Σ, A) :

4.31 Definition (Representation of Inequations): A *representation of a Σ -inequation* $\alpha \leq \gamma : U \rightarrow V$ is any representation R of Σ such that $\alpha^R \leq \gamma^R$ pointwise, i.e. such that for any set X and any $y \in U(R)(X)$,

$$\alpha_X^R(y) \leq \gamma_X^R(y) .$$

We say that such a representation R *satisfies* the inequation $\alpha \leq \gamma$.

For a set A of Σ -inequations, we call *representation of (Σ, A)* any representation of Σ that satisfies each inequation of A . We define the category of representations of the 2-signature (Σ, A) to be the full subcategory $\text{Rep}^\Delta(\Sigma, A)$ of the category of representations of Σ whose objects are representations of (Σ, A) .

4.32 Example (Ex. 4.28 continued): We denote by β the Σ_{ULC} -inequation

$$\text{app} \circ (\text{abs} \times \text{id}) \leq \text{subst} \quad . \quad (\beta)$$

We write $(\Sigma_{\text{ULC}}, \beta) := (\Sigma_{\text{ULC}}, \{\beta\})$. A representation P of $(\Sigma_{\text{ULC}}, \beta)$ is given by

- a monad $P : \text{Set} \xrightarrow{\Delta} \text{Pre}$ and
- two morphisms of P -modules

$$\text{app} : P \times P \rightarrow P \quad \text{and} \quad \text{abs} : P' \rightarrow P$$

such that for any set X and any $y \in P(X')$ and $z \in PX$

$$\text{app}_X(\text{abs}_X(y), z) \leq y[* := z] \quad .$$

4.4. Initiality for 2-Signatures

Given a 2-signature (Σ, A) , we would like to exhibit an initial object in its associated category of representations of (Σ, A) . However, we have to rule out inequations which are never satisfied, since an empty category obviously does not have an initial object. We restrict ourselves to inequations with a classic codomain:

4.33 Definition (Classic Inequation): A Σ -inequation is *classic* if its codomain is classic.

4.34 Theorem: For any set of classic Σ -inequations A , the category of representations of (Σ, A) has an initial object.

Proof. The basic ingredients for building the initial representation are given by the initial representation $\Delta \hat{\Sigma}$ in the category $\text{Rep}^\Delta(\Sigma)$ (cf. Lem. 4.21) or, equivalently, by the initial representation $\hat{\Sigma}$ in $\text{Rep}(\Sigma)$. We call $\hat{\Sigma}$ the monad underlying the representation $\hat{\Sigma}$.

The proof consists of three steps: at first, we define a preorder \leq_A on the terms of $\hat{\Sigma}$, induced by the set A of inequations. Afterwards we show that the data of the representation $\hat{\Sigma}$ — substitution, representation morphisms etc. — is compatible with the preorder \leq_A in a suitable sense. This will yield a representation $\hat{\Sigma}_A$ of (Σ, A) . Finally we show that $\hat{\Sigma}_A$ is the initial such representation.

— *The monad underlying the initial representation:*

For any set X , we equip $\hat{\Sigma}X$ with a preorder A by setting, for $x, y \in \hat{\Sigma}X$,

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \text{Rep}^\Delta(\Sigma, A), \quad i_R(x) \leq_R i_R(y) \quad , \quad (4.4.1)$$

where $i_R : \Delta \hat{\Sigma} \rightarrow R$ is the initial morphism of representations of Σ , cf. Lem. 4.21. We have to show that the map

$$X \mapsto \hat{\Sigma}_A X := (\hat{\Sigma}X, \leq_A)$$

4. Reductions for Untyped Syntax

yields a relative monad on Δ . The missing fact to prove is that the substitution with a morphism

$$f \in \text{Pre}(\Delta X, \hat{\Sigma}_A Y) \cong \text{Set}(X, \hat{\Sigma} Y)$$

is compatible with the order \leq_A : given any $f \in \text{Pre}(\Delta X, \hat{\Sigma}_A Y)$ we show that $\sigma^{\hat{\Sigma}}(f) : \text{Set}(\hat{\Sigma} X, \hat{\Sigma} Y)$ is monotone with respect to \leq_A and hence (the carrier of) a morphism $\sigma(f) : \text{Pre}(\hat{\Sigma}_A X, \hat{\Sigma}_A Y)$. We overload the infix symbol $\gg=$ to denote monadic substitution. Suppose $x \leq_A y$, we show

$$x \gg= f \leq_A y \gg= f .$$

Using the definition of \leq_A , we must show, for any representation R of (Σ, A) ,

$$i_R(x \gg= f) \leq_R i_R(y \gg= f) .$$

Since i_R is a morphism of representations, it is compatible with the substitution of $\hat{\Sigma}$ and U_*R ; we have

$$i_R(x \gg= f) = i_R(x) \gg= i_R \circ f .$$

Rewriting this equality and its equivalent for y in the current goal yields the goal

$$i_R(x) \gg= i_R \circ f \leq_A i_R(y) \gg= i_R \circ f ,$$

which is true since the substitution of R (whose underlying map is that of U_*R) is monotone in the first argument (cf. [Rem. 2.86](#)) and $i_R(x) \leq_R i_R(y)$ by assumption. We hence have defined a monad $\hat{\Sigma}_A$ over Δ . We interrupt the proof for an important lemma:

4.35 Lemma: *Given a classic Σ -module $V : \text{Rep}^\Delta(\Sigma) \rightarrow \text{LMod}(\Delta, \text{wPre})$ from the category of representations of Σ in monads on Δ to the large category of modules over such monads, we have*

$$x \leq_A y \in V(\hat{\Sigma})(X) \iff \forall R : \text{Rep}^\Delta(\Sigma, A), \quad V(i_R)(x) \leq_{V_X^R} V(i_R)(y) ,$$

where now and later we omit the argument X , e.g., in $V(i_R)(X)(x)$.

Proof of Lem. 4.35. The proof is done by induction on the derivation of “ V classic”. The only interesting case is where $V = M \times N$ is a product:

$$\begin{aligned} (x_1, y_1) \leq (x_2, y_2) &\iff x_1 \leq x_2 \wedge y_1 \leq y_2 \\ &\iff \forall R, M(i_R)(x_1) \leq M(i_R)(x_2) \wedge \forall R, N(i_R)(y_1) \leq N(i_R)(y_2) \\ &\iff \forall R, M(i_R)(x_1) \leq M(i_R)(x_2) \wedge N(i_R)(y_1) \leq N(i_R)(y_2) \\ &\iff \forall R, V(i_R)(x_1, y_1) \leq V(i_R)(x_2, y_2) . \end{aligned}$$

□

— *Representing Σ in $\hat{\Sigma}_A$:*

Any arity $s \in \Sigma$ should be represented by the module morphism $s^{\hat{\Sigma}}$, i.e. by the representation of s in $\hat{\Sigma}$. We have to show that those representations are compatible with the preorder \leq_A . Given $x \leq_A y$ in $\text{dom}(s, \hat{\Sigma})(X)$, we show (omitting the argument X in $s^{\hat{\Sigma}}(X)(x)$)

$$s^{\hat{\Sigma}}(x) \leq_A s^{\hat{\Sigma}}(y) .$$

By definition, we have to show that, for any representation R as before,

$$i_R(s^{\hat{\Sigma}}(x)) \leq_R i_R(s^{\hat{\Sigma}}(y)) .$$

Since i_R is a morphism of representations, it commutes with the representational module morphisms — the corresponding diagram is similar to the diagram of [Def. 4.17](#). By rewriting with this equality we obtain the goal

$$s^R((\text{dom}(s)(i_R))(x)) \leq_R s^R((\text{dom}(s)(i_R))(y)) .$$

This goal is proved by instantiating [Lem. 4.35](#) with the classic Σ -module $\text{dom}(s)$ (cf. [Def. 4.29](#)) and the fact that s^R is monotone. We hence have established a representation — which we call $\hat{\Sigma}_A$ — of Σ in the monad $\hat{\Sigma}_A$.

— $\hat{\Sigma}_A$ satisfies A :

The next step is to show that the representation $\hat{\Sigma}_A$ satisfies A . Given an inequation

$$\alpha \leq \gamma : U \rightarrow V$$

of A with a classic Σ -module V , we must show that for any set X and any $x \in U(\hat{\Sigma}_A)(X)$ in the domain of α we have

$$\alpha_X^{\hat{\Sigma}_A}(x) \leq_A \gamma_X^{\hat{\Sigma}_A}(x) . \quad (4.4.2)$$

In the following we omit the subscript X . By [Lem. 4.35](#) the goal is equivalent to

$$\forall R : \text{Rep}^\Delta(\Sigma, A), \quad V(i_R)(\alpha^{\hat{\Sigma}_A}(x)) \leq_{V_X^R} V(i_R)(\gamma^{\hat{\Sigma}_A}(x)) . \quad (4.4.3)$$

Let R be a representation of (Σ, A) . We continue by proving [Disp. \(4.4.3\)](#) for R . By [Rem. 4.24](#) and the fact that i_R is also the carrier of a morphism of representations of Σ from $\Delta\hat{\Sigma}$ to R (cf. [Lem. 4.20](#)) we can rewrite the goal as

$$\alpha^R(U(i_R)(x)) \leq_{V_X^R} \gamma^R(U(i_R)(x)) ,$$

which is true since R satisfies A .

— *Initiality of $\hat{\Sigma}_A$:*

Given any representation R of (Σ, A) , the morphism i_R is monotone with respect to the preorders on $\hat{\Sigma}_A$ and R by construction of \leq_A . It is hence a morphism of representations from $\hat{\Sigma}_A$ to R . Uniqueness of the morphisms i_R follows from its uniqueness in the category of representations of Σ , i.e. without inequations. Hence $\hat{\Sigma}_A$ is the initial object in the category of representations of (Σ, A) . \square

4. Reductions for Untyped Syntax

4.36 Remark: Note that the proof of the main theorem uses the equivalence proved in [Lem. 4.35](#) in *both* directions. The implication from left to right would be ensured automatically if we had defined Σ -modules to be functors into the category $\text{LRMod}(\Delta, \text{Pre})$ instead of $\text{LRMod}(\Delta, \text{wPre})$. See [Rem. 4.23](#) for an explanation why we still choose the latter category as codomain category.

4.37 Remark: Note that for a classic Σ -module V we can actually prove the implication from left to right of [Lem. 4.35](#) more generally: for *any* morphism of representations $f : P \rightarrow R$ (not just an initial one as in [Lem. 4.35](#)) the module morphism $V(f) : V(P) \rightarrow V(R)$ is monotone. Again the only interesting case is where $V = V_1 \times V_2$ is a product. Let X be a set and $x = (x_1, x_2)$ and $y = (y_1, y_2)$ in $V(P)(X)$:

$$\begin{aligned} (x_1, x_2) \leq_{V^P} (y_1, y_2) &\Leftrightarrow x_1 \leq_{V_1^P} y_1 \wedge x_2 \leq_{V_2^P} y_2 \\ &\Rightarrow V_1(f)(x_1) \leq_{V_1^R} V_1(f)(y_1) \wedge V_2(f)(x_2) \leq_{V_2^R} V_2(f)(y_2) \\ &\Leftrightarrow (V_1(f)(x_1), V_2(f)(x_2)) \leq_{V^R} (V_1(f)(y_1), V_2(f)(y_2)) \\ &\Leftrightarrow V(f)(x_1, x_2) \leq_{V^R} V(f)(y_1, y_2) . \end{aligned}$$

4.38 Example ([Ex. 4.32](#) continued): The only inequation [Disp. \(\$\beta\$ \)](#) of the signature $(\Sigma_{\text{ULC}}, \beta)$ is classic. The initial representation of $(\Sigma_{\text{ULC}}, \beta)$ is given by the monad ULC_β together with the ULC_β -module morphisms Abs and App (cf. [Ex. 2.95](#)) as representation structure.

We conclude this section with some remarks about “generating inequalities”, (regular) monads and *fully faithful* morphisms:

4.39 Remark about “Generating” Inequations: Given a 2-signature (Σ, A) and a representation R of Σ , the representation morphism of modules s^R of any $s \in \Sigma$ of R is monotone. For the initial representation of (Σ, A) this means that any relation between terms of Σ which comes from A is automatically propagated into subterms. Similarly, the relation on those terms is by construction reflexive and transitive, since we consider representations in monads with codomain Pre .

For the example of ULC_β this means that in order to obtain a complete reduction relation, it is sufficient to enforce only one rule by an inequation, which is

$$(\lambda M)N \leq M[* := N] .$$

4.40 Remark about Finite Contexts: Altenkirch et al. [[ACU10](#)] characterize the untyped lambda calculus as a relative monad on the inclusion functor $i : \text{Fin} \rightarrow \text{Set}$ from finite sets to sets. An anonymous referee suggested combining our viewpoint — syntax as monad over $\Delta : \text{Set} \rightarrow \text{Pre}$ — with Altenkirch et al.’s one might consider the lambda calculus as a relative monad on the composition $\Delta \circ i : \text{Fin} \rightarrow \text{Pre}$, and, more generally, one might consider representations of a signature (Σ, A) over monads on $\Delta \circ i : \text{Fin} \rightarrow \text{Pre}$.

The above theorem remains true when replacing monads on Δ by monads on $\Delta \circ i$ everywhere. An equivalence between the theorem thus obtained and our [Thm. 4.34](#) might be established in a way similar to what Zsidó [[Zsi10](#)] does in her PhD thesis: she shows, by means of adjunctions between the respective categories of models, the equivalence between the approach of Fiore et al. [[FPT99](#)] — based on monoids over finite contexts — and the approach of Hirschowitz and Maggesi [[HM07a](#)], where models are built from monads on the category \mathbf{Set} , i.e. over arbitrary contexts.

4.41 Remark about Monads on \mathbf{Pre} : As mentioned in [Sect. 1.5](#), Ghani and L  th [[GL03](#)] and Hirschowitz and Maggesi [[HM10a](#)] suggest the use of monads over the category \mathbf{Pre} of preordered sets for modelling syntax with a rewriting relation. Indeed, representations of a signature (Σ, A) could be analogously defined for such monads. The above construction of the initial representation of (Σ, A) carries over to representations in such monads, thus yielding an initiality result in which syntax is modelled as monad on \mathbf{Pre} . It might be interesting to establish a precise connection — e.g., in form of adjunctions — between the resulting categories of representations in monads on \mathbf{Pre} and representations in relative monads on Δ .

4.42 Remark about Fully Faithful Translations: By construction any morphism $f : P \rightarrow Q$ of representations of a 2-signature (Σ, A) is *faithful*, i.e. it sends related terms $x \rightsquigarrow y$ in $P(X)$ to related terms $f_X(x) \rightsquigarrow f_X(y)$ in $Q(X)$. It is natural to ask whether f is also *full*, that is, whether each $f_X : P(X) \rightarrow Q(X)$ is a full functor between the preorders $P(X)$ and $Q(X)$, considered as functors. Explicitly, this means to ask whether for any $x, y \in P(X)$ such that $f_X(x) \rightsquigarrow f_X(y)$ in $Q(X)$ we have $x \rightsquigarrow y$.

5. SIMPLE TYPE SYSTEMS WITH REDUCTIONS

This chapter aims to combine the contents of [Chapts. 3](#) and [4](#) in order to obtain an initiality result for simple type systems with reductions on the term level. This result thus accounts for our example from [Sect. 1.1](#): the translation from PCF with its usual reduction relation to the untyped lambda calculus with beta reduction. The goal thus is to define a notion of *signature* and suitable *representations* for such signatures, such that the types and terms generated by the signature, equipped with reductions according to the inequations specified by the signature, form the *initial representation*. Analogously to the previous chapter, we define a notion of *2-signature* with two levels: a *syntactic* level specifying types and terms of a language, and, on top of that, a *semantic* level specifying reduction rules on the terms.

5.1. 1-Signatures

From the *syntactic* point of view presented in [Sect. 3.3.1.1](#), 1-signatures for types and terms are the same as in [Chapt. 3](#), [Def. 3.46](#). We have to adapt the *semantic* definition of signatures for terms, however, since we now work with relative monads on Δ^T for some set T instead of monads over families of sets. The following definition is the analogue of [Def. 3.34](#), adapted to the use of relative monads:

5.1 Definition (Relative S -Monad): Given an algebraic signature S , the *category S -RMnd* of *relative S -monads* is defined as the category whose objects are pairs (T, P) of a representation T of S and a relative monad

$$P : \text{Set}^T \xrightarrow{\Delta^T} \text{Pre}^T .$$

A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S -representations $g : T \rightarrow T'$ and a morphism of relative monads $f : P \rightarrow P'$ over the retyping functor \vec{g} as in [Rem. 2.89](#).

Given $n \in \mathbb{N}$, we write $S\text{-RMnd}_n$ for the category whose objects are pairs (T, P) of a representation T of S and a relative monad P over Δ_n^T . A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S -representations $g : T \rightarrow T'$ and a monad morphism $f : P \rightarrow P'$ over the retyping functor $\vec{g}(n)$ defined in [Def. 2.28](#).

Similarly, we have a large category of modules over relative monads:

5. Simple Type Systems with Reductions

5.2 Definition (Large Category $\text{LRMod}_n(S, \mathcal{D})$ of Modules): Given a natural number $n \in \mathbb{N}$, an algebraic signature S and a category \mathcal{D} , we call $\text{LRMod}_n(S, \mathcal{D})$ the category an object of which is a pair (P, M) of a relative S -monad $P \in S\text{-RMnd}_n$ and a P -module with codomain \mathcal{D} . A morphism to another such (Q, N) is a pair (f, h) of a morphism of relative S -monads $f : P \rightarrow Q$ in $S\text{-RMnd}_n$ and a morphism of relative modules $h : M \rightarrow f^*N$.

As before, we sometimes just write the module — i.e. the second — component of an object or morphism of the large category of modules. Given $M \in \text{LRMod}_n(S, \mathcal{D})$, we thus write $M(V)$ or M_V for the value of the module on the object V .

A *half-arity over S of degree n* is a functor from relative S -monads to the category of large modules of degree n :

5.3 Definition (Half-Arity over S (of degree n)): Given an algebraic signature S and $n \in \mathbb{N}$, we call *half-arity over S of degree n* a functor

$$\alpha : S\text{-RMnd} \rightarrow \text{LRMod}_n(S, \text{Pre}) .$$

which is pre-inverse to the forgetful functor.

As before we restrict ourselves to a class of such functors. Again, we start with the *tautological* module:

5.4 Definition (Tautological Module of Degree n): Given $n \in \mathbb{N}$, any relative monad R over Δ^T induces a monad R_n over Δ_n^T with object map $(V, t_1, \dots, t_n) \mapsto (RV, t_1, \dots, t_n)$. To any relative S -monad R we associate the tautological module of R_n ,

$$\Theta_n(R) := (R_n, R_n) \in \text{LRMod}_n(S, \text{Pre}_n^T) .$$

Furthermore, we again use *canonical natural transformations* (cf. [Def. 3.41](#)) to build *classic* half-arities; these transformations specify context extension (derivation) and selection of specific object types (fibre):

5.5 Definition (Classic Half-Arity): As with monads (cf. [Sect. 3.3](#)), we restrict our attention to *classic* half-arities, which we define analogously to [Def. 3.42](#) as constructed using derivations and products, starting from the fibres of the tautological module and the constant singleton module. We omit the precise statement of this definition.

A half-arity of degree n thus associates, to any relative S -monad P over a set of types T , a family of P -modules indexed by T^n :

5.6 Remark *Module of Higher Degree corresponds to a Family of Modules (II)*: [Remark 3.37](#) applies analogously to modules over relative modules. More precisely, let T be a set and let R be a monad on the functor Δ^T . Then a module M over the monad R_n corresponds precisely to a family of R -modules $(M_t)_{t \in T^n}$ by (un)currying. Similarly, a morphism $\alpha : M \rightarrow N$ of modules of degree n is equivalent to a family $(\alpha_t)_{t \in T^n}$ of morphisms of modules of degree zero with $\alpha_t : M_t \rightarrow N_t$.

An arity of degree $n \in \mathbb{N}$ for terms over an algebraic signature S is defined to be a pair of functors from relative S -monads to modules in $\text{LRMod}_n(S, \text{Pre})$. The degree n corresponds to the number of object type indices of its associated constructor. As an example, the arities of `Abs` and `App` of [Ex. 1.3](#) are of degree 2.

5.7 Definition (Term-Arity, Signature over S): A *classic arity* α over S of degree n is a pair

$$s = (\text{dom}(\alpha), \text{cod}(\alpha))$$

of half-arithies over S of degree n such that

- $\text{dom}(\alpha)$ is classic and
- $\text{cod}(\alpha)$ is of the form $[\Theta_n]_\tau$ for some canonical natural transformation τ as in [Def. 3.41](#).

Any classic arity is thus *syntactically* of the form given in [Disp. \(3.3.5\)](#). Note, however, that the definition of Θ in [Sect. 3.3](#) differs from the one used in the present chapter. We write $\text{dom}(\alpha) \rightarrow \text{cod}(\alpha)$ for the arity α , and $\text{dom}(\alpha, R) := \text{dom}(\alpha)(R)$ and similar for the codomain and morphisms of relative S -monads. Given a weighted set (J, d) as in [Def. 3.44](#), a term-signature Σ over S indexed by (J, d) is a J -family Σ of classic arities over S , the arity $\Sigma(j)$ being of degree $d(j)$ for any $j \in J$.

5.8 Definition (Typed Signature): A *typed signature* is a pair (S, Σ) consisting of an algebraic signature S for sorts and a term-signature Σ (indexed by some weighted set) over S .

5.9 Example: [Ex. 3.47](#) and [3.48](#) still apply. Note, however, that the underlying definition of Θ differs from that of [Sec. 3](#), and that fibre and derivation are adapted accordingly.

5.2. Representations of 1-Signatures

5.10 Definition (Representation of an Arity, a Signature over S): A representation of an arity α over S in an S -monad R is a morphism of relative modules

$$\text{dom}(\alpha, R) \rightarrow \text{cod}(\alpha, R) .$$

A representation R of a signature over S is given by a relative S -monad — called R as well — and a representation α^R of each arity α of S in R .

Representations of (S, Σ) are the objects of a category $\text{Rep}^\Delta(S, \Sigma)$, whose morphisms are defined as follows:

5. Simple Type Systems with Reductions

5.11 Definition (Morphism of Representations): Given representations P and R of a typed signature (S, Σ) , a morphism of representations $f : P \rightarrow R$ is given by a morphism of relative S -monads $f : P \rightarrow R$, such that for any arity α of Σ the following diagram of module morphisms commutes:

$$\begin{array}{ccc} \text{dom}(\alpha, P) & \xrightarrow{\alpha^P} & \text{cod}(\alpha, P) \\ \text{dom}(\alpha, f) \downarrow & & \downarrow \text{cod}(\alpha, f) \\ \text{dom}(\alpha, R) & \xrightarrow{\alpha^R} & \text{cod}(\alpha, R). \end{array}$$

5.12 Lemma: For any typed signature (S, Σ) , the category of representations of (S, Σ) has an initial object.

Proof. The initial object is obtained, analogously to the untyped case (cf. Lem. 4.5, 4.20, 4.21), via an adjunction $\Delta_* \dashv U_*$ between the categories of representations of (S, Σ) in relative monads and those in monads as in Chapt. 3.

In more detail, to any relative S -monad $(T, P) \in S\text{-RMnd}$ we associate the S -monad $U(T, P) := (T, UP)$ where U_*P is the monad obtained by postcomposing with the forgetful functor $U^T : \text{Pre}^T \rightarrow \text{Set}^T$. Substitution for U_*P is defined, in each fibre, as in Lem. 2.83. For any arity $s \in \Sigma$ we have that

$$U_* \text{dom}(s, P) \cong \text{dom}(s, U_*P) ,$$

and similar for the codomain. The postcomposed representation morphism $U_*s(P)$ hence represents s in U_*P in the sense of Chapt. 3. This defines the functor $U_* : \text{Rep}^\Delta(S, \Sigma) \rightarrow \text{Rep}(S, \Sigma)$. Conversely, to any S -monad we can associate a relative S -monad by postcomposing with $\Delta^T : \text{Set}^T \rightarrow \text{Pre}^T$, analogous to the untyped case in Def. 4.15, yielding $\Delta_* : \text{Rep}(S, \Sigma) \rightarrow \text{Rep}^\Delta(S, \Sigma)$. In summary, the natural isomorphism

$$\varphi_{R,P} : (\text{Rep}^\Delta(S, \Sigma))(\Delta_*R, P) \cong (\text{Rep}(S, \Sigma))(R, U_*P)$$

is given by postcomposition with the forgetful functor (from left to right) resp. the functor Δ (from right to left). □

5.3. Inequations

Analogously to the untyped case (cf. Defs. 4.22, 4.30), an inequation associates, to any representation of (S, Σ) in a relative monad P , two parallel morphisms of P -modules. However, similarly to arities, an inequation may now be, more precisely, a *family of inequations*, indexed by object types. Consider the simply-typed lambda calculus, which

was defined with *typed* abstraction and application. Similarly, we have a *typed substitution* operation for TLC, which substitutes a term of type $s \in T_{\text{TLC}}$ for a free variable of type s in a term of type $t \in T_{\text{TLC}}$, yielding again a term of type t . For $s, t \in T_{\text{TLC}}$ and $M \in \text{TLC}(V^{*s})_t$ and $N \in \text{TLC}(V)_s$, beta reduction is specified by

$$\lambda_{s,t} M(N) \rightsquigarrow M[* := N] ,$$

where our notation hides the fact that not only abstraction, but also application and substitution are typed operations. More formally, such a reduction rule might read as a family of inequations between morphisms of modules

$$\text{app}_{s,t} \circ (\text{abs}_{s,t} \times \text{id}) \leq _ [*^s :=_t _] ,$$

where $s, t \in T_{\text{TLC}}$ range over types of the simply-typed lambda calculus. Analogously to Sect. 3.3, we want to specify the beta rule without referring to the set T_{TLC} , but instead express it for an arbitrary representation R of the typed signature $(S_{\text{TLC}}, \Sigma_{\text{TLC}})$ (cf. Exs. 3.2, 3.47), as in

$$\text{app}^R \circ (\text{abs}^R \times \text{id}) \leq _ [* := _] ,$$

where both the left and the right side of the inequation are given by suitable R -module morphisms of degree 2. Source and target of a *half-equation* accordingly are given by functors from representations of a typed signature (S, Σ) to a suitable category of modules. A half-equation then is a natural transformation between its source and target functor:

5.13 Definition (Category of Half-Equations): Let (S, Σ) be a signature. An (S, Σ) -module U of degree $n \in \mathbb{N}$ is a functor from the category of representations of (S, Σ) as defined in Sect. 5.2 to the category $\text{LRMod}_n(S, \text{wPre})$ (cf. Def. 5.2) commuting with the forgetful functor to the category of relative monads. We define a morphism of (S, Σ) -modules to be a natural transformation which becomes the identity when composed with the forgetful functor. We call these morphisms *half-equations* (of degree n). We write $U^R := U(R)$ for the image of the representation R under the S -module U , and similar for morphisms.

5.14 Definition (Substitution as Half-Equation): Given a relative monad on Δ^T , its associated substitution-of-one-variable operation (cf. Def. 2.111) yields a family of module morphisms, indexed by pairs $(s, t) \in T$. By Rem. 5.6 this family is equivalent to a module morphism of degree 2. The assignment

$$\text{subst} : R \mapsto \text{subst}^R : [\hat{R}_2]_2^1 \times [\hat{R}_2]_1 \rightarrow [\hat{R}_2]_2$$

thus yields a half-equation of degree 2 over any signature S . Its domain and codomain are classic.

5.15 Example (Ex. 3.47 continued): The map

$$\text{app} \circ (\text{abs} \times \text{id}) : R \mapsto \text{app}^R \circ (\text{abs}^R \times \text{id}^R) : [\hat{R}_2]_2^1 \times [\hat{R}_2]_1 \rightarrow [\hat{R}_2]_2$$

is a half-equation over the signature TLC, as well as over the signature of PCF.

5.16 Definition: Any classic arity of degree n ,

$$s = [\Theta_n]_{\sigma_1}^{\tau_1} \times \dots \times [\Theta_n]_{\sigma_m}^{\tau_m} \rightarrow [\Theta_n]_{\sigma} ,$$

defines a classic S -module

$$\text{dom}(s) : R \mapsto [R_n]_{\sigma_1}^{\tau_1} \times \dots \times [R_n]_{\sigma_m}^{\tau_m} .$$

5.17 Definition (Inequation): Given a signature (S, Σ) , an *inequation over* (S, Σ) , or (S, Σ) -*inequation*, of degree $n \in \mathbb{N}$ is a pair of parallel half-equations between (S, Σ) -modules of degree n . We write $\alpha \leq \gamma$ for the inequation (α, γ) . We leave the degree implicit whenever possible, analogously to [Rem. 3.33](#).

5.18 Example (Beta Reduction): For any suitable 1-signature — i.e. for any 1-signature that has an arity for abstraction and an arity for application — we specify beta reduction through an inequation of degree 2 using the parallel half-equations of [Def. 5.14](#) and [Ex. 5.15](#):

$$\text{app} \circ (\text{abs} \times \text{id}) \leq \text{subst} : [\Theta]_2^1 \times [\Theta]_1 \rightarrow [\Theta]_2 .$$

5.19 Example (Fixpoints and Arithmetics of PCF): The reduction rules for PCF are informally given in [Fig. A.4](#). We specify these reduction rules as inequations over the 1-signature of PCF (cf. [Ex. 3.48](#)) as follows:

$$\begin{aligned} \text{app} \circ (\text{abs} \times \text{id}) &\leq \text{subst} : [\Theta]_2^1 \times [\Theta]_1 \rightarrow [\Theta]_2 \\ \text{Fix} &\leq \text{app} \circ (\text{id}, \text{Fix}) : [\Theta]_{1 \Rightarrow 1} \rightarrow [\Theta]_1 \\ \text{app} \circ (\text{Succ}, \mathbf{n}) &\leq \mathbf{n} + \mathbf{1} : * \rightarrow [\Theta]_t \\ \text{app} \circ (\text{Pred}, \mathbf{0}) &\leq \mathbf{0} : * \rightarrow [\Theta]_t \\ \text{app} \circ (\text{Pred}, \text{app} \circ (\text{Succ}, \mathbf{n})) &\leq \mathbf{n} : * \rightarrow [\Theta]_t \\ \text{app} \circ (\text{Zero?}, \mathbf{0}) &\leq \mathbf{T} : * \rightarrow [\Theta]_o \\ \text{app} \circ (\text{Zero?}, \text{app} \circ (\text{Succ}, \mathbf{n})) &\leq \mathbf{F} : * \rightarrow [\Theta]_o \\ &\vdots \end{aligned}$$

5.20 Definition (Representation of Inequations): A *representation of an* (S, Σ) -*inequation* $\alpha \leq \gamma : U \rightarrow V$ (of degree n) is any representation R over a set of types T of (S, Σ) such that $\alpha^R \leq \gamma^R$ pointwise, i.e. if for any pointed context $(X, \mathbf{t}) \in \text{Set}^T \times T^n$, any $t \in T$ and any $y \in U_{(X, \mathbf{t})}^R(t)$,

$$\alpha^R(y) \leq \gamma^R(y) , \tag{5.3.1}$$

where we omit the sort argument t as well as the context (X, \mathbf{t}) from α and γ . We say that such a representation R *satisfies* the inequation $\alpha \leq \gamma$.

For a set A of (S, Σ) -inequations, we call *representation of $((S, \Sigma), A)$* any representation of (S, Σ) that satisfies each inequation of A . We define the category of representations of the 2-signature $((S, \Sigma), A)$ to be the full subcategory of the category of representations of S whose objects are representations of $((S, \Sigma), A)$. We also write (Σ, A) for $((S, \Sigma), A)$.

According to [Rem. 5.6](#), the inequation of [Disp. \(5.3.1\)](#) is equivalent to ask whether, for any $\mathbf{t} \in T^n$, any $t \in T$ and any $y \in U_{\mathbf{t}}^R(X)(t)$,

$$\alpha_{\mathbf{t}}^R(y) \leq \gamma_{\mathbf{t}}^R(y) .$$

5.4. Initiality for 2-Signatures

We are ready to state and prove an initiality result for typed signatures with inequations:

5.21 Theorem: *For any set of classic (S, Σ) -inequations A , the category of representations of $((S, \Sigma), A)$ has an initial object.*

Proof. The proof is analogous to that of the untyped case (c.f. [Thm. 4.34](#)). The fact that we now consider *typed* syntax introduces a minor complication, on the presentation of which we put the emphasis during the proof. The basic ingredients for building the initial representation are given by the initial representation $(\hat{S}, \hat{\Sigma})$ — or just $\hat{\Sigma}$ for short — in the category $\text{Rep}(S, \Sigma)$ of representations in monads on set families (cf. [Thm. 3.53](#)). Equivalently, the ingredients come from the initial object $(\hat{S}, \Delta_* \hat{\Sigma})$ — or just $\Delta_* \hat{\Sigma}$ for short — of representations *without inequations* in the category $\text{Rep}^\Delta(S, \Sigma)$ (cf. [Lem. 5.12](#)). We call $\hat{\Sigma}$ resp. $\Delta_* \hat{\Sigma}$ the monad resp. relative monad underlying the initial representation

The proof consists of 3 steps: at first, we define a preorder \leq_A on the terms of $\hat{\Sigma}$, induced by the set A of inequations. Afterwards we show that the data of the representation $\hat{\Sigma}$ — substitution, representation morphisms etc. — is compatible with the preorder \leq_A in a suitable sense. This will yield a representation $\hat{\Sigma}_A$ of (Σ, A) . Finally we show that $\hat{\Sigma}_A$ is the initial such representation.

— *The monad underlying the initial representation:*

For any context $X \in \text{Set}^{\hat{S}}$ and $t \in \hat{S}$, we equip $\hat{\Sigma}X(t)$ with a preorder A by setting — *morally*, cf. below —, for $x, y \in \hat{\Sigma}X(t)$,

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \text{Rep}(\Sigma, A), \quad i_R(x) \leq_R i_R(y) , \quad (5.4.1)$$

where $i_R : \Delta_* \hat{\Sigma} \rightarrow R$ is the initial morphism of representations of (S, Σ) , cf. [Lem. 5.12](#). Note that the above definition in [Disp. \(5.4.1\)](#) is ill-typed: we have $x \in \hat{\Sigma}X(t)$, which cannot be applied to (a fibre of) $i_R(X) : \vec{g}(\hat{\Sigma}X) \rightarrow R(\vec{g}X)$. We denote by $\varphi = \varphi_R$ the natural isomorphism induced by the adjunction of [Rem. 2.23](#) and [Def. 2.22](#) obtained by

5. Simple Type Systems with Reductions

retyping — along the initial morphism of types $g : \hat{S} \rightarrow T = T_R$ — towards the set T of “types” of R ,

$$\varphi_{X,Y} : \text{Pre}^T(\vec{g}(\hat{\Sigma}X), R(\vec{g}X)) \cong \text{Pre}^{\hat{S}}(\hat{\Sigma}X, R(\vec{g}X) \circ g) .$$

Instead of the above definition in [Disp. \(5.4.1\)](#), we should really write

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \text{Rep}(\Sigma, A), \quad (\varphi(i_{R,X}))(x) \leq_R (\varphi(i_{R,X}))(y) , \quad (5.4.2)$$

where we omit the subscript “ R ” from φ . We have to show that the map

$$X \mapsto \hat{\Sigma}_A X := (\hat{\Sigma}X, \leq_A)$$

yields a relative monad on $\Delta^{\hat{S}}$. The missing fact to prove is that the substitution with a morphism

$$f \in \text{Pre}^{\hat{S}}(\Delta X, \hat{\Sigma}_A Y) \cong \text{Set}^{\hat{S}}(X, \hat{\Sigma}Y)$$

is compatible with the order \leq_A : given any $f \in \text{Pre}^{\hat{S}}(\Delta X, \hat{\Sigma}_A Y)$ we show that

$$\sigma^{\hat{\Sigma}}(f) \in \text{Set}^{\hat{S}}(\hat{\Sigma}X, \hat{\Sigma}Y)$$

is monotone with respect to \leq_A and hence (the carrier of) a morphism

$$\sigma^{\hat{\Sigma}_A}(f) \in \text{Pre}^{\hat{S}}(\hat{\Sigma}_A X, \hat{\Sigma}_A Y) .$$

We overload the infix symbol $\gg=$ to denote monadic substitution. Note that this notation now hides an implicit argument giving the sort of the term in which we substitute. Suppose $x, y \in \hat{\Sigma}X(t)$ with $x \leq_A y$, we show

$$x \gg= f \leq_A y \gg= f .$$

Using the definition of \leq_A , we must show, for a given representation R of (Σ, A) ,

$$(\varphi(i_R))(x \gg= f) \leq_R (\varphi(i_R))(y \gg= f) . \quad (5.4.3)$$

Let g be the initial morphism of types towards the types of R . Since $i := i_R$ is a morphism of representations — and thus in particular a *monad* morphism, it is compatible with the substitution of $\hat{\Sigma}$ and R ; we have

$$\begin{array}{ccc} \vec{g}(\hat{\Sigma}X) & \xrightarrow{\vec{g}(\sigma(f))} & \vec{g}(\hat{\Sigma}Y) \\ \downarrow i_X & & \downarrow i_Y \\ R(\vec{g}X) & \xrightarrow{\sigma^R(i_Y \circ \vec{g}f)} & R(\vec{g}Y) . \end{array} \quad (5.4.4)$$

By applying the isomorphism φ on the diagram of [Disp. \(5.4.4\)](#), we obtain

$$\begin{aligned}\varphi(i_Y) \circ \sigma(f) &= \varphi(i_Y \circ \vec{g}(\sigma(f))) \\ &= \varphi(\sigma(i_Y \circ \vec{g}f) \circ i_X) \\ &= g^*(\sigma^R(i_Y \circ \vec{g}f)) \circ \varphi(i_X) .\end{aligned}\tag{5.4.5}$$

Rewriting the equality of [Disp. \(5.4.5\)](#) twice in the goal [Disp. \(5.4.3\)](#) yields the goal

$$g^*(\sigma^R(i_Y \circ \vec{g}f))((\varphi(i_X))(x)) = g^*(\sigma^R(i_Y \circ \vec{g}f))((\varphi(i_X))(y)) ,$$

which is true since $g^*(\sigma^R(i_Y \circ \vec{g}f))$ is monotone and $(\varphi(i_X))(x) \leq_R (\varphi(i_X))(y)$ by hypothesis. We hence have defined a monad $\hat{\Sigma}_A$ over $\Delta^{\hat{S}}$.

5.22 Lemma: *[Lemma 4.35](#) generalizes to the typed setting of this chapter.*

Proof of [Lem. 5.22](#). The proof is analogous to the proof of [Lem. 4.35](#): we apply the same reasoning in the corresponding fibre. □

— *Representing Σ in $\hat{\Sigma}_A$:*

Any arity $s \in \Sigma$ should be represented by the module morphism $s^{\hat{\Sigma}}$, i.e. by the representation of s in $\hat{\Sigma}$. We have to show that those representations are compatible with the preorder A . Given $x \leq_A y$ in $\text{dom}(s, \hat{\Sigma})(X)$, we show (omitting the argument X in $s^{\hat{\Sigma}}(X)(x)$)

$$s^{\hat{\Sigma}}(x) \leq_A s^{\hat{\Sigma}}(y) .$$

By definition, we have to show that, for any representation R with initial morphism $i = i_R : \hat{\Sigma} \rightarrow R$ as before,

$$\varphi(i_X)(s^{\hat{\Sigma}}(x)) \leq_R \varphi(i_X)(s^{\hat{\Sigma}}(y)) .$$

But these two sides are precisely the images of x and y under the upper-right composition of the diagram of [Def. 5.11](#) for the morphism of representations i_R . By rewriting with this diagram we obtain the goal

$$s^R((\text{dom}(s)(i_R))(x)) \leq_R s^R((\text{dom}(s)(i_R))(y)) .$$

We know that s^R is monotone, thus it is sufficient to show

$$(\text{dom}(s)(i_R))(x) \leq_R (\text{dom}(s)(i_R))(y) .$$

This goal follows from [Lem. 5.22](#) (instantiated for the classic S -module $\text{dom}(s)$, cf. [Def. 5.16](#)) and the hypothesis $x \leq_A y$. We hence have established a representation — which we call $\hat{\Sigma}_A$ — of S in $\hat{\Sigma}_A$.

5. Simple Type Systems with Reductions

— $\hat{\Sigma}_A$ satisfies A :

The next step is to show that the representation $\hat{\Sigma}_A$ satisfies A . Given an inequation

$$\alpha \leq \gamma : U \rightarrow V$$

of A with a classic S -module V , we must show that for any context $X \in \text{Set}^{\hat{S}}$, any $t \in \hat{S}$ and any $x \in U(\hat{\Sigma}_A)(X)_t$ in the domain of α we have

$$\alpha^{\hat{\Sigma}_A}(x) \leq_A \gamma^{\hat{\Sigma}_A}(x) ,$$

where here and later we omit the context argument X and the sort argument t . By [Lem. 5.22](#) the goal is equivalent to

$$\forall R : \text{Rep}(\Sigma, A), \quad V(i_R)(\alpha^{\hat{\Sigma}_A}(x)) \leq_{V_X^R} V(i_R)(\gamma^{\hat{\Sigma}_A}(x)) . \quad (5.4.6)$$

Let R be a representation of (Σ, A) . We continue by proving [Disp. \(5.4.6\)](#) for R . [Remark 4.24](#) holds analogously in the typed setting of this chapter. The fact that i_R is the carrier of a morphism of (S, Σ) -representations from $\Delta \hat{\Sigma}$ to R allows to rewrite the goal as

$$\alpha^R(U(i_R)(x)) \leq_{V_X^R} \gamma^R(U(i_R)(x)) ,$$

which is true since R satisfies A .

— *Initiality of $\hat{\Sigma}_A$:*

Given any representation R of (Σ, A) , the morphism i_R is monotone with respect to the orders on $\hat{\Sigma}_A$ and R by construction of \leq_A . It is hence a morphism of representations from $\hat{\Sigma}_A$ to R . Uniqueness of the morphisms i_R follows from its uniqueness in the category of representations of (S, Σ) , i.e. without inequations. Hence $(\hat{S}, \hat{\Sigma}_A)$ is the initial object in the category of representations of $((S, \Sigma), A)$. □

5.23 Remark *Iteration Principle by Initiality:* The universal property of the language generated by a 2-signature yields an *iteration principle* to define maps — translations — on this language, which are certified to be compatible with substitution and reduction in the source and target languages. How does this iteration principle work? More precisely, what data (and proof) needs to be specified in order to define such a translation via initiality from a language, say, $(\hat{S}, \hat{\Sigma}_A)$ to another language $(\hat{S}', \hat{\Sigma}'_{A'})$, generated by signatures (S, Σ, A) and (S', Σ', A') , respectively? The translation is a morphism — an initial one — in the category of representations of the signature (S, Σ, A) of the source language. It is obtained by equipping the relative monad $\hat{\Sigma}'_{A'}$ underlying the target language with a representation of the signature (S, Σ, A) . In more detail:

1. we give a representation of the type signature S in the set \hat{S}' . By initiality of \hat{S} , this yields a translation $\hat{S} \rightarrow \hat{S}'$ of sorts.

2. Afterwards, we specify a representation of the term signature Σ in the monad $\hat{\Sigma}'_{A'}$ by defining suitable (families) of morphisms of $\hat{\Sigma}'_{A'}$ -modules. This yields a representation R of (S, Σ) in the monad $\hat{\Sigma}'_{A'}$.

By initiality, we obtain a morphism $f : (\hat{S}, \hat{\Sigma}) \rightarrow R$ of representations of (S, Σ) , that is, we obtain a translation from $(\hat{S}, \hat{\Sigma})$ to $(\hat{S}', \hat{\Sigma}')$ as the colax monad morphism underlying the morphism f . However, we have not yet ensured that the translation f is compatible with the respective reduction preorders in the source and target languages.

3. Finally, we verify that the representation R of (S, Σ) satisfies the inequations of A , that is, we check whether, for each $\alpha \leq \gamma : U \rightarrow V \in A$, and for each context V , each $t \in \hat{S}$ and $x \in U_V^R(t)$,

$$\alpha^R(x) \leq \gamma^R(x) .$$

After verifying that R satisfies the inequations of A , the representation R is in fact a representation of (S, Σ, A) . The initial morphism f thus yields a faithful translation from $(\hat{S}, \hat{\Sigma}_A)$ to $(\hat{S}', \hat{\Sigma}'_{A'})$.

5.24 Example (Translation from PCF to ULC, [Exs. 3.54](#) and [5.19](#) cont.): Recall the translations from PCF to the untyped lambda calculus of [Ex. 3.54](#). We might attempt to specify the same translations using the iteration operator obtained by [Thm. 5.21](#), where PCF is equipped with the reduction relation generated by the inequations of [Ex. 5.19](#) and ULC is equipped with beta reduction as in [Ex. 4.32](#). However, representing the fixedpoint operator of PCF by the lambda term Θ fails, for reasons explained at the end of [Chapt. 9](#).

For the translation of PCF to the lambda calculus mapping the fixedpoint operator of PCF to the Turing fixedpoint combinator, we have formalized its specification via initiality in the proof assistant Coq [[Coq10](#)]. After constructing the category of representations of PCF, we equip the untyped lambda calculus with a representations of PCF, representing the arity **Fix** by the Turing operator Θ . The formalization is explained in [Chapt. 9](#). Note that the translation is given by a Coq function and hence executable.

PART II.

COMPUTER IMPLEMENTATION

6. FORMALIZING CATEGORY THEORY IN Coq

In this chapter we describe our computer formalization of general concepts of category theory as presented in [Chapt. 2](#). We start with a brief introduction to our favourite theorem prover Coq [[Coq10](#)]. We then describe the challenges one encounters when one attempts to formalize algebraic structures in general, and category theory in particular, in Coq. Finally we present our implementation of monads and modules over monads as well as their relative counterparts. Throughout the chapter we explain features of Coq when we first encounter them.

6.1. About the Proof Assistant Coq

The proof assistant Coq [[Coq10](#)] is an implementation of the *Calculus of Inductive Constructions (CIC)* which itself is a constructive *type theory*. Its objects are *terms* built according to a grammar (see the Coq manual [[The10](#)] for the term forming rules). Each valid term has its associated *type* which is itself a term and which is automatically computed by Coq. In Coq a typing judgment is written $t : T$, meaning that t is a term of type T . Typing judgments are for example $1 : \text{Nat}$ and $\text{plus} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$. Function application is simply denoted by a blank, i.e. we write $f\ x$ for $f(x)$.

The CIC also treats propositions as types via the *Curry–Howard isomorphism*, hence a proof of a proposition P is in fact a term of type P . Accordingly, a proof of a proposition $A \Rightarrow B$ is a function $A \rightarrow B$, i.e. a term which associates a proof of B to any proof of A . As an example, the function $\text{id} : P \rightarrow P$ is a proof of the tautology $P \Rightarrow P$. In the proof assistant Coq a user hence proves a proposition P by providing a term p of type P . Coq checks the validity of the proof p by checking whether $p : P$.

Coq comes with extensive support to *interactively build* the proof terms of a given proposition. In *proof mode* so-called *tactics* help the users to reduce the proposition they want to prove — the *goal* — into one or more simpler subgoals, until reaching trivial subgoals which can be solved directly.

Particular concepts of Coq such as records and type classes, setoids, implicit arguments and coercions are explained in a call-by-need fashion in the course of the thesis. One important feature is the [Section](#) mechanism (cf. also the Coq manual [[The10](#)]). Parameters and hypotheses declared in a section automatically get discharged when closing the section. Constants of the section then become functions, depending on an argument of the type of the parameter they mentioned. We illustrate this concept by means of a small

example; consider the following Coq declarations:

```
Section def_double.  
Variable n : nat.  
Definition double : nat := 2 * n.  
Check double.  
double  
      : nat  
Print Assumptions double.  
Section Variables:  
n : nat
```

Inside the section `def_double`, the constant `double` is of type `nat`, as we verify using the `Check` command. Furthermore, it depends on the section variable `n : nat` declared using the `Variable` vernacular command. After closing the section, the constant `double` is a closed term of function type:

```
End def_double.  
Check double.  
double  
      : nat -> nat  
Print Assumptions double.  
Closed under the global context  
Eval compute in double 4.  
= 8 : nat
```

In our formalization, we use the `Section` mechanism extensively. When presenting a definition depending on section variables, we either give a slightly modified, fully discharged version of the statement — compared to the actual Coq code —, or mention the section variables informally in the text.

6.2. Formalizing Algebraic Structures

An algebraic structure typically is given by some data — i.e. sets and operations on them — that satisfies given properties. For instance, a group is given by a set, together with a binary associative multiplication and a unit element, such that any element of the set has a multiplicative inverse. Such algebraic structures are defined in a *hierarchic* way: for instance, any *group* is a particular *monoid* that admits inverses. Thus any group is a monoid. The other way round, given a group, if multiplication is commutative, then this group is actually abelian, and the group is an element of the class of abelian groups.

This hierarchic structure poses a major problem in the formalization of classic mathematics, and the question of how to formalize algebraic structures is a subject of active

research. Put simply, the main question is how tightly one should pack together the data and properties of an algebraic structure. If data and properties are packed together tightly, then operations and properties can easily be associated to their respective underlying sets, and this allows for overloading notation and coercions. On the other hand, this tight packing makes it difficult to “add” data and properties to an instance of an algebraic structure, e.g., to consider a group, for which one has proved commutativity of multiplication, as an abelian group. We do not attempt to propose a solution to the challenge of how to formalize algebraic structures. However, we need to choose from the existing solutions. In Coq there are basically two possible answers: *records*, employed e.g., by Garillot et al. [GGMR09], correspond to a tight packing of algebraic structure, whereas *type classes* [SO08], as used by Spitters and v. d. Weegen [SvdW11], correspond to a rather loose packaging.

Coq records are implemented as an inductive data type with one constructor. However, use of the vernacular command `Record` (instead of plain `Inductive`) allows the optional automatic definition of the projection functions to the constructor arguments – the “fields” of the record. Additionally, one can declare those projections as *coercions*, i.e. they can be inserted automatically by Coq, and left out in printing. As an example for a coercion, it allows us to write $c : C$ for an object c of a category C . Here the projection from the category type to the type of objects of a category is declared as a coercion (cf. Code 6.1). This is the formal counterpart to the convention introduced in the informal definition of categories in Def. 2.1. Another example of coercion is given in the definition of monad (cf. Def. 2.33), where it corresponds precisely to the there-mentioned *abuse of notation*. Finally, an example of coercion that is *not* given by a projection is given by the tautological module, i.e. the map that associates to any monad P the tautological P -module (cf. Def. 2.48).

Type classes are implemented as records. Similarly to the difference between records and inductive types, type classes are distinguished from records only in that some meta-theoretic features are automatically enabled when declaring an algebraic structure as a class rather than a record. For details we refer to Sozeau’s article about the implementation of type classes [SO08] and Spitters and v. d. Weegen’s work [SvdW11]. Type classes differ from records in their usage, more specifically, in which data one declares as a *parameter* of the structure and which one declares as a *field*. The following example, borrowed from [SvdW11], illustrates the different uses; we give two definitions of the algebraic structure of reflexive relation, one in terms of classes and one in terms of records:

```
Class Reflexive {A : Type}{R : relation A} :=
  reflexive : forall a, R a a.
```

```
Record Reflexive := {
  carrier : Type ;
```

```
car_rel : relation carrier ;
rel_refl : forall a, car_rel a a }.
```

Our main interest in classes comes from the fact that by using classes many of the arguments of projections are automatically declared as *implicit arguments*. This leads to more readable code since arguments that can be deduced by Coq do not have to be written down. Thus it corresponds precisely to the mathematical practice of not mentioning arguments (e.g. indices) which “are clear from the context”. An instance of this behaviour can be seen in the definition of category in [Def. 2.1](#), where we omit the 3 “object” arguments — written as an index — of the dependent composition of morphisms. In particular, the structure argument of the projection, that is, the argument specifying the instance whose field we want to access, is implicit and deduced automatically by Coq. This mechanism allows for *overloading*, a prime example being the implementation of setoids (cf. [Sect. 6.3.3](#)) as a type class; in a term “ $a == b$ ” denoting setoidal equality, Coq automatically finds the correct setoid instance from the type of a and b ¹.

We decide to define our algebraic structures in terms of type classes first, and bundle the class together with some of the class parameters in a record afterwards, as is shown in the following example for the type class `Cat_struct` (cf. [Code 6.3](#)) and the bundling record `Cat`.

6.1 Code (Bundling a type class into a record):

```
Record Cat := {
  obj:> Type ;
  mor: obj -> obj -> Type ;
  cat_struct:> Cat_struct mor }.
```

This duplication of Coq definitions is a burden rather than a feature. We still proceed like this for the following reasons: in our case the use of records is unavoidable since we want to have a Coq *type* of categories, of functors between two given categories, etc. This is necessary when those objects — functors, for instance — shall themselves be the objects or morphisms of some category, as is clear from [Code 6.1](#). However, we profit from aforementioned features of type classes, notably automatic declaration of some arguments as *implicit* and the resulting overloading.

Apart from that, we do not employ any feature that makes the use of type classes comfortable — such as maximally inserted arguments, operational classes, etc. — since we usually work with the bundled versions. Readers who are interested in how to use type classes in Coq properly, are advised to take a look at Spitters and v. d. Weegen’s paper [\[SvdW11\]](#). There, the authors employ the mentioned bundling of type classes in

¹Beware! In case several instances of setoid have been declared on one and the same Coq type, the instance chosen by Coq might not be the one intended by the user. This is the main reason for Spitters and v. d. Weegen to restrict the fields of type classes to *propositions*.

records only when necessary, e.g., when the considered structures are to be the objects or morphisms of some category.

6.3. Formalizing Categories

As seen in [Sect. 2.1](#), there are two definitions of category ([Def. 2.1](#), [Rem. 2.3](#)), which are equivalent from the point of view of a mathematician. When implementing category theory in dependent type theory, however, one needs to choose the one or the other definition. This section explains how we implement categories in Coq and some consequences of our design choice.

6.3.1. Which Definition to Formalize — Dependent Hom-Sets?

The main difference concerning formalization between these two definitions is that of *composability of morphisms*. The first definition can be implemented directly only in type theories featuring *dependent types*, such as the Calculus of Inductive Constructions (CIC). The ambient type system, i.e. the prover, then takes care of composability – terms with compositions of non-composable morphisms are rejected as ill-typed terms.

The second definition can be implemented also in provers with a simpler type system such as the family of HOL theorem provers. However, since those (as well as the CIC) are theories where functions are total, one is left with the question of how to implement composition. Composition might then be implemented either as a functional relation or as a total function about which nothing is known (deducible) on non-composable morphisms. The second possibility is implemented in O’Keefe’s library [[O’K04](#)]. There the author also gives an overview of available formalizations in different theorem provers with particular attention to the choice of the definition of category.

In our favourite prover Coq, both definitions have been employed in significant developments: the second definition is used in Simpson’s construction of the Gabriel–Zisman localization [[Sim06](#)], whereas Huet and Saïbi’s ConCaT [[HS98](#)] uses type families of morphisms as in [Def. 2.1](#). To our knowledge there is no library in a prover with dependent types such as Coq or NuPRL [[CAA⁺86](#)] which develops and compares both definitions with respect to provability, readability, and other criteria.

We decided to construct our library using type families of morphisms. In this way the proof of composability of two morphisms is done by Coq type computation automatically. As a consequence, we are sometimes obliged to insert trivial isomorphisms between equal — but not convertible — objects of some category, in order to make compositions typecheck. For an example see [Sect. 7.1.2](#).

Coq’s *implicit argument* mechanism allows us to omit the deducible arguments, as we do in [Def. 2.1](#) for the “object arguments” c, d and e of the composition. Together with

the possibility to define infix notations, this brings our formal syntax close to informal mathematical syntax.

6.3.2. Setoidal Equality on Morphisms

All the properties of a category \mathcal{C} concern equality of two parallel morphisms, i.e. morphisms with same source and target. In Coq there is a polymorphic equality, called *Leibniz equality*, readily available for any type. However, this equality actually denotes *syntactic equality*, which already in the case of maps does not coincide with the “mathematical” equality on maps – given by pointwise equality – that we would rather consider. With the use of axioms — for the mentioned example of maps the axiom `functional_extensionality` from the Coq standard library — one can often deduce Leibniz equality from the “mathematical equality” in question. But this easily gets cumbersome, in particular when the morphisms — as will be in our case — are sophisticated algebraic structures composed of a lot of data and properties. Instead, we require any collection of morphisms $\mathcal{C}(c, d)$ for objects c and d of \mathcal{C} to be equipped with an equivalence relation, which plays the rôle of equality on this collection. In the Coq standard library equivalence relations are implemented as a type class with the underlying type as a parameter A , and the relation as well as a proof of it being an equivalence as fields:

6.2 Code (Setoid Type Class):

```
Class Setoid A := {
  equiv : relation A ;
  setoid_equiv :> Equivalence equiv }.
```

Setoids as morphisms of a category have been used by Aczel [Acz93] in LEGO (there a setoid is simply called “set”) and Huet and Saïbi (HS) [HS98] in Coq. HS’s setoids are implemented as records of which the underlying type is a component instead of a parameter. This choice makes it necessary to duplicate the definitions of setoids and categories in order to make them available with a “higher” type ².

6.3.3. Coq Setoids and Setoid Morphisms

Setoids in Coq are implemented as a type class (cf. Code 6.2) with a type parameter A and a relation on A as well as a proof of this relation being an equivalence as fields. For the term `equiv a b` the infix notation “`a == b`” is introduced. The instance argument of `equiv` is implicit (cf. Sect. 6.2).

²In HS’s `CONCAT`, a type T that is defined after the type of setoids cannot be the carrier of a setoid itself. As a remedy, HS define a type `Setoid'` isomorphic to `Setoid` after the definition of T . The type of `Setoid'` now being higher than that of T , one can define a term of type `Setoid'` whose carrier is T .

A *morphism of setoids* between setoids A and B is a Coq function on the underlying types which is compatible with the setoid relations on the source and target. That is, it maps equivalent terms of A to equivalent terms of B , or, in mathematical notation,

$$a \equiv_A a' \text{ implies } f(a) \equiv_B f(a') . \quad (6.3.1)$$

In the Coq standard library such morphisms are implemented as a type class

```
Class Proper {A} (R : relation A) (m : A) : Prop :=
  proper_prf : R m m.
```

where the type A is instantiated with a function type $A \rightarrow B$ and the relation R on $A \rightarrow B$ is instantiated with pointwise compatibility³:

```
Definition respectful (A B : Type) (R : relation A) (R' : relation B) :=
  fun f g => forall x y, R x y -> R' (f x) (g y).
Notation "R ==> R'" := (@respectful _ _ (R%signature) (R'%signature))
(right associativity, at level 55) : signature_scope.
```

Given Coq types A and B equipped with relations $R : \text{relation } A$ and $R' : \text{relation } B$, respectively, and a map $f : A \rightarrow B$, the statement $\text{Proper } (R ==> R') f$ — replacing aforementioned notation — really means

$\text{Proper } (\text{respectful } R \ R') f$,

which is the same as $\text{respectful } R \ R' f f$, which itself just means

$\text{forall } x y, R x y \rightarrow R' (f x) (f y)$.

This is indeed the statement of [Disp. \(6.3.1\)](#) in the special case that R and R' are *equivalence* relations.

For any component of an algebraic structure that is a map defined on setoids, we add a condition of the form $\text{Proper} \dots$ in the formalization. Examples are the categorical composition ([Code 6.3](#)) and the monadic substitution map ([Code 6.10](#)). Rewriting related terms under those equivalence relations is tightly integrated in the [rewrite](#) tactic of Coq.

6.3.4. Coq Implementation of Categories

As a result of the aforementioned considerations, we adopt Sozeau's definition of category [[SO08](#)], which itself is a variant of the definition given by Huet and Saïbi [[HS98](#)]. Unlike Huet and Saïbi's contribution ConCaT, Sozeau's approach uses *type classes* for algebraic structures and thus avoids the universe inconsistencies that have to be circumvented by duplicating definitions in ConCaT (cf. [Sect. 6.3.2](#)). More precisely, in Sozeau's implementation of setoids (cf. [Code 6.2](#)), the carrier type is a *parameter* instead of a *field* as in

³ In the Coq standard library the definition of `respectful` is actually a special case of a more general definition of a heterogeneous relation `respectful_hetero`.

Huet and Saïbi's. Our type class of categories is parametrized by a type of objects and a type family of morphisms, whose parameters are the source and target objects.

6.3 Code (Type Class of Categories):

```
Class Cat_struct (obj : Type)(mor : obj -> obj -> Type) := {
  mor_oid :> forall a b, Setoid (mor a b) ;
  id : forall a, mor a a ;
  comp : forall {a b c}, mor a b -> mor b c -> mor a c ;
  comp_oid :> forall a b c, Proper (equiv ==> equiv ==> equiv) (@comp a b c)
  ;
  id_r : forall a b (f: mor a b), comp f (id b) == f ;
  id_l : forall a b (f: mor a b), comp (id a) f == f ;
  assoc : forall a b c d (f: mor a b) (g: mor b c) (h: mor c d),
    comp (comp f g) h == comp f (comp g h) }.

```

Compared to [Def. 2.1](#) there are two additional fields: the field

`mor_oid :> forall a b, Setoid (mor a b)`

equips each collection of morphisms `mor a b` with a custom equivalence relation. The field `comp_oid` states that the composition `comp` of the category is compatible with the setoidal structure on the morphisms given by the field `mor_oid` as explained in [Sect. 6.3.3](#). We recall that setoidal equality is overloaded and denoted by the infix symbol `'=='`. In the following we write `'a ----> b'` for `mor a b` and `f;g` for the composition of morphisms `f : a ----> b` and `g : b ----> c`⁴.

6.3.5. The Categories of Interest

The category `Set` is formalized in Coq as the category of Coq types. By using Coq types and functions as objects and morphisms of this category, we obtain executable Coq substitution and translation maps, cf. [Code 9.11](#).

6.4 Code (Set, [Def. 2.4](#)):

```
Program Instance TYPE_struct : Cat_struct (fun a b => a -> b) := {
  mor_oid a b := TYPE_hom_oid a b ;
  id a := fun x : a => x ;
  comp a b c := fun (f : a -> b) (g : b -> c) => fun x => g (f x) }.

```

In this instance declaration, the fields `id_r`, `id_l` and `assoc` are filled automatically by the [Program](#) framework, cf. [Sect. 6.3.7](#). For a set T , the category Set^T of [Def. 2.20](#) has,

⁴Coq deduces and inserts the missing “object” arguments `a`, `b` and `c` of the composition automatically from the type of the morphisms. For this reason those arguments are called *implicit* (cf. [Sect. 6.2](#)).

as objects, Coq type families indexed by T . Morphisms between two such objects are suitable families of Coq functions :

6.5 Code (Category of Type Families):

```
Program Instance ITYPE_struct : Cat_struct (obj := T -> Type)
  (fun A B => forall t, A t -> B t) := {
  mor_oid := INDEXED_TYPE_oid ; (* pointwise equal. in each component *)

  comp A B C f g := fun t => fun x => g t (f t x) ;
  id A := fun t x => x }.
```

We also employ categories whose objects are families of *preordered* sets (i.e. Coq types), and morphisms are monotone Coq functions. We omit their definition.

6.3.6. Initial Objects

Initial objects have been defined in [Def. 2.5](#). Formally, we implement the initiality structure as a type class, parametrized by categories. Its fields are given by an object `Init` of the category, a map `InitMor` mapping each object `a` of the category to a morphism from `Init` to `a` and a proposition stating that `InitMor a` is unique for any object `a`.

```
Class Initial (C : Cat) := {
  Init : C ;
  InitMor: forall a : C, Init ----> a ;
  InitMorUnique: forall a (f : Init ----> a), f == InitMor a }.
```

Note that the initial morphism is *not* given by an existential statement of the form $\forall a, \exists f : \dots$, or, in Coq terms, using an `exists` statement. This is because the Coq existential lies in [Prop](#) and hence does not allow for elimination — witness extraction — when building anything but proofs.

6.3.7. Interlude on the **Program** feature

The **Program Instance** vernacular allows to fill in fields of an instance of a type class by means of tactics. Indeed, when omitting a field in an instance declaration — such as the proofs of associativity `assoc` and left and right identity `id_l` and `id_r` in [Code 6.4](#). — the **Program** framework creates an *obligation* for each missing field, making use of the information that the user provided for the other fields. As an example, the obligation created for the field `assoc` of the previous example is to prove associativity for the composition defined by

```
comp f g := fun x => g (f x) .
```

It then tries to solve the resulting obligations using the tactic that the user has specified via the **Obligation Tactic** command. In case the automatic resolution of the obligation fails, the user can enter the interactive proof mode finish the proof manually.

It is technically possible to fill in both *data* and *proof* fields automatically via the **Program** framework. However, in order to avoid the automatic inference of data which we cannot control, we always specify *data* directly as is done in [Code 6.4](#), and rely on automation via **Program** only for *proofs*.

6.3.8. Retyping and Option

We present the formalization of some commonly used definitions. The reader might want to skip this section and come back to it when being pointed back here.

We define retyping (cf. [Rem. 2.23](#)) for families of sets and preordered sets through an inductive type:

6.6 Code (Retyping Functor, [Rem. 2.23](#)):

```
Variables (T T' : Type) (g : T -> T').
Inductive retype (V : ITYPE T) : ITYPE T' :=
| ctype : forall t, V t -> retype V (g t).
```

The constructor `ctype : forall V t, V t -> retype V (g t)` is the carrier of the natural transformation of the same name of [Rem. 2.23](#). Given a family V of *preordered* sets, the preorder on $\vec{g}V$ is induced by the preorder on V :

```
Inductive retype_ord (V : IPO T) : forall u, relation (retype g V (u)) :=
| ctype_ord : forall t (x y : V t), x <<< y
  -> retype_ord (ctype g x) (ctype g y).
```

The option data type is implemented in the module `Coq.Init.Datatypes` of the Coq standard library.

6.7 Code (Option, [Sect. 2.2.3.1](#)):

```
Inductive option (A:Type) : Type :=
| Some : A -> option A
| None : option A.
```

We can turn the map $A \mapsto A' := A + \{*\}$ into a monad as follows:

6.8 Code (Option Monad):

```
Program Instance option_monad_s :
  Monad_struct (C:=TYPE) (option) := {
  weta := @Some ;
  kleisli a b f := fun t => match t with
```

```

    | Some y => f y
    | None => None
  end }.

```

There is also a typed variant of the option data type:

6.9 Code (Typed Option, [Sect. 2.2.3.1](#)):

```

Inductive opt (u : T) (V : ITYPE T) : ITYPE T :=
  | some : forall t : T, V t -> opt u V t
  | none : opt u V u.

```

Given a list l over T , the multiple addition of variables with (object language) types according to l to a set of variables V is defined by recursion over l . For this enriched set of variables we introduce the notation $V * * l$.

```

Fixpoint pow (l : [T]) (V : ITYPE T) : ITYPE T :=
  match l with
  | nil => V
  | b::bs => pow bs (opt b V)
  end.

```

The map `opt` is functorial, as is the multiple addition of variables `pow`. On morphisms the `pow` operation is defined by recursively applying the functoriality of `opt`, where for the latter we use a special notation with a prefixed hat.

```

Fixpoint pow_map (l : [T]) V W (f : V ----> W) :
  V * * l ----> W * * l :=
  match l return V * * l ----> W * * l with
  | nil => f
  | b::bs => pow_map (^f)
  end.

```

6.4. Monads, Modules and their Morphisms

Implementing monads leaves one with the choice between the definitions given in [Def. 2.33](#) and [Def. 2.65](#). The first definition, while preferred by category theorists, has the inconvenience that defining instances of monads such as monadic syntax would require proving commutativity of the square, e.g., using multiple induction for monadic syntax. Furthermore the second definition is well-known in the programming community for its use in `HASKELL`. We thus decide to implement the definition of [Def. 2.65](#). Since we are mainly interested in its instances over the category of (families of) sets, we can define convenient infix notation for its substitution.

6. Formalizing Category Theory in Coq

Formally, a monad (cf. [Def. 2.65](#)) is a type class parametrized by a category C and a function $F : C \rightarrow C$ on the objects of C :

6.10 Code (Monad, [Def. 2.65](#)):

```
Class Monad_struct (C : Cat) (F : C -> C) := {
  weta : forall c, c ----> (F c);
  kleisli : forall a b, (a ----> F b) -> (F a ----> F b);
  kleisli_oid :> forall a b, Proper (equiv ==> equiv) (kleisli (a:=a) (b:=b));
  eta_kl : forall a b (f : a ----> F b), weta a ;; kleisli f == f;
  kl_eta : forall a, kleisli (weta a) == id _;
  dist : forall a b c (f : a ----> F b) (g : b ----> F c),
    kleisli f ;; kleisli g == kleisli (f ;; kleisli g) }.
```

Monads admit a functorial structure:

6.11 Code (Functoriality for Monads, [Rem. 2.66](#)):

Variable $T : \text{Monad } C$.

Definition lift : forall a b (f : a ----> b), T a ----> T b :=
 fun a b f => kleisli (f ;; weta b).

We present two different implementations of monad morphisms. The more general definition implements colax monad morphisms as defined in [Def. 2.69](#):

6.12 Code (Colax Monad Morphism, [Def. 2.69](#)):

```
Class colax_Monad_Hom_struct (Tau : forall c, F (P c) ----> Q (F c)) := {
  gen_monad_hom_kl : forall c d (f : c ----> P d),
    #F (kleisli f) ;; Tau _ ==
    Tau _ ;; (kleisli (#F f ;; Tau _)) ;
  gen_monad_hom_weta : forall c : C,
    #F (weta c) ;; Tau _ == weta _ }.
```

When working exclusively with a special case of a more general definition, it is more convenient to implement this special case as a separate definition: for two monads P and Q over the same category C , a *simple morphism of monads* — as used in [Sect. 3.2](#) — is given by a family of morphisms $\tau_c \in \mathcal{C}(Pc, Qc)$ that is compatible with the monadic structure:

6.13 Code (Simple Monad Morphism, [Def. 3.12](#)):

```
Class Monad_Hom_struct (Tau : forall c, P c ----> Q c) := {
  monad_hom_kl : forall c d (f : c ----> P d),
    kleisli f ;; Tau d == Tau c ;; kleisli (f ;; Tau d) ;
  monad_hom_weta : forall c : C, weta c ;; Tau c == weta c }.
```


It follows from these commutativity properties that the family τ is a natural transformation between the functors induced by the monads P and Q . Given a monad P over \mathcal{C} , a P -module with codomain D is formalized as follows:

6.14 Code (Module, [Def. 2.71](#)):

Variable $P : \text{Monad } C$.

Class `Module_struct` ($M : C \rightarrow D$) := {
`mkleisli`: `forall` $c\ d$, $(c \dashrightarrow P\ d) \rightarrow (M\ c \dashrightarrow M\ d)$;
`mkleisli_oid`: \rightarrow `forall` $c\ d$, `Proper` (`equiv` ==> `equiv`) (`mkleisli` ($c:=c$)($d:=d$));
`mkl_weta`: `forall` c , `mkleisli` (`weta` c) == `id` $_$;
`mkl_mkl`: `forall` $c\ d\ e$ ($f : c \dashrightarrow P\ d$) ($g : d \dashrightarrow P\ e$),
 `mkleisli` f ;; `mkleisli` g == `mkleisli` (f ;; `kleisli` g) }.

For two modules S and T with codomain \mathcal{D} over a monad P as above, a module morphism from S to T is given by a family of maps, indexed by the objects of \mathcal{C} , commuting with module substitution:

6.15 Code (Module Morphism, [Def. 2.73](#)):

Class `Module_Hom_struct` (N : `forall` x , $S\ x \dashrightarrow T\ x$) := {
`mod_hom_mkl`: `forall` $c\ d$ ($f : c \dashrightarrow P\ d$),
 `mkleisli` f ;; $N\ _$ == $N\ _$;; `mkleisli` f }.

6.5. Relative Monads, Formalized

As opposed to (plain) monads, we have only one definition of relative monads available. The implementation of this definition in Coq is similar to that of monads (cf. [Code 6.10](#)). Given a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, a relative monad is given by a map $T : \mathcal{C} \rightarrow \mathcal{D}$ on the objects of the categories involved, and data analogous to that of a monad:

6.16 Code (Relative Monad, [Def. 2.75](#)):

Variables $C\ D : \text{Cat}$.

Variable $F : \text{Functor } C\ D$.

Class `RMonad_struct` ($T : C \rightarrow D$) := {
`rweta`: `forall` $c : C$, $F\ c \dashrightarrow T\ c$;
`rkleisli`: `forall` $a\ b : C$, $F\ a \dashrightarrow T\ b \rightarrow T\ a \dashrightarrow T\ b$;
`rkleisli_oid`: \rightarrow `forall` $a\ b$, `Proper` (`equiv` ==> `equiv`) (`rkleisli` ($a:=a$) ($b:=b$)) ;
`reta_kl`: `forall` $a\ b$: `obC`, `forall` $f : F\ a \dashrightarrow T\ b$, `rweta` a ;; `rkleisli` f == f ;
`rkl_eta`: `forall` a , `rkleisli` (`rweta` a) == `id` $_$;
`rdist`: `forall` $a\ b\ c$ ($f : F\ a \dashrightarrow T\ b$) ($g : F\ b \dashrightarrow T\ c$),
 `rkleisli` f ;; `rkleisli` g == `rkleisli` (f ;; `rkleisli` g) }.

Analogously to monads we define functoriality for a given relative monad P :

6.17 Code (Functoriality for Relative Monads, [Rem. 2.80](#)):

Variable $P : \text{RMonad } C$.

Definition $\text{rlift} : \text{forall } a \ b \ (f : a \multimap b), P \ a \multimap P \ b :=$
 $\text{fun } a \ b \ f \Rightarrow \text{rkleisli } (\#F \ f \ ; \ \text{rweta } b).$

In the following we consider morphisms of relative monads in varying generality: one definition ([Def. 4.3](#)) is analogous to the simple morphisms of monads (cf. [Code 6.13](#)), another implements the colax version of [Def. 2.87](#). For the statement of the second, general, definition, we place ourselves in the environment given in [Def. 2.87](#). In short, we have a natural transformation $N : F'G \Rightarrow G'F : \mathcal{C} \rightarrow \mathcal{D}'$.

6.18 Code (Colax Morphism of Relative Monads, [Def. 2.87](#)):

Variable $N : \text{NT } (\text{CompF } G \ F') \ (\text{CompF } F \ G').$

Class $\text{colax_RMonad_Hom_struct } (\text{tau} : \text{forall } c : C, G' \ (P \ c) \multimap Q \ (G \ c)) := \{$
 $\text{gen_rmonad_hom_rweta} : \text{forall } c : C,$
 $\quad N _ \ ; \ \#G' \ (\text{rweta } c) \ ; \ \text{tau } c == \text{rweta } (G \ c) \ ;$
 $\text{gen_rmonad_hom_rkl} : \text{forall } (c \ d : C) \ (f : F \ c \multimap P \ d),$
 $\quad \#G' \ (\text{rkleisli } f) \ ; \ \text{tau } d == \text{tau } c \ ; \ \text{rkleisli } (a := G \ c) \ (N \ c \ ; \ \#G' \ f \ ; \ \text{tau } _) \}.$

A *module* M over a relative monad P (on a functor F) is given by data similar to that of a module over a monad, except for the insertion of applications of F where necessary.

6.19 Code (Module over a Relative Monad, [Def. 2.90](#)):

Class $\text{RModule_struct } (M : C \multimap E) := \{$
 $\text{rmkleisli} : \text{forall } c \ d \ (f : F \ c \multimap P \ d), M \ c \multimap M \ d \ ;$
 $\text{rmkleisli_oid} :> \text{forall } c \ d, \text{Proper } (\text{equiv} ==> \text{equiv}) \ (\text{rmkleisli } (c := c) (d := d)) \ ;$
 $\text{rmkl_rweta} : \text{forall } c : C, \text{rmkleisli } (\text{rweta } c) == \text{id } (M \ c) \ ;$
 $\text{rmkl_rmkl} : \text{forall } c \ d \ e \ (f : F \ c \multimap P \ d) \ (g : F \ d \multimap P \ e),$
 $\quad \text{rmkleisli } f \ ; \ \text{rmkleisli } g == \text{rmkleisli } (f \ ; \ \text{rkleisli } g) \}.$

Given two modules M and N with codomain \mathcal{D} over a relative monad P , a module morphism from M to N is given by a collection of maps $(S_c : M \ c \rightarrow N \ c)_{c \in \mathcal{C}}$ commuting with module substitution:

6.20 Code (Morphism of Relative Modules, [Def. 2.94](#)):

Variables $M \ N : \text{RModule } P \ D$.

Class $\text{RModule_Hom_struct } (S : \text{forall } c : C, M \ c \multimap N \ c) := \{$
 $\text{rmod_hom_rmkl} : \text{forall } c \ d \ (f : F \ c \multimap P \ d),$
 $\quad \text{rmkleisli } f \ ; \ S \ d == S \ c \ ; \ \text{rmkleisli } f \}.$

7. FORMALIZATION OF ZSIDÓ'S THEOREM

In this chapter we describe the formalization in the proof assistant Coq [Coq10] of Zsidó's initiality theorem presented in Sect. 3.2. In particular, we explain what we omitted in the informal presentation — the construction of the initial representation of a given simply-typed signature.

7.1. Signatures & Representations

An *arity* determines the type and binding behaviour of a *constructor*, and a *signature* is a family of arities. A *representation* of a signature S is given by a monad P (over a suitable category) and a morphism of P -modules for each arity α of S , where the source and target module of this morphism are determined by α . Among those representations the object of interest is the *initial* one, i.e. the representation from which there exists exactly one *morphism of representations* to any other representation. The initial representation is called *syntax generated by S* .

7.1.1. Using Lists for Algebraic Arities & Signatures

For the formal definitions let us fix a set T of object language types. As explained in Def. 3.20, an algebraic arity over T is determined by a pair of a list of data and an element $t_0 \in T$, yielding an efficient and concise way to specify algebraic arities. An algebraic signature could thus be implemented — as in Def. 3.9 — as a pair consisting of a type `sig_index` — which is used for indexing the arities — and a map from the indexing type to the actual arity type, which is simply built using lists — for which we employ a Haskell-like notation — and products:

7.1 Code (Signature, Def. 3.9):

Notation `"[T]"` := (list `T`) (at level 5).

Record `Signature` : `Type` := {
 `sig_index` : `Type`;
 `sig` : `sig_index` \rightarrow `[[T] * T] * T` }.

However, a slight modification turns out to be useful. During the construction of the initial representation, a universal quantification over arities of a signature S with a

given target type $t \in T$ is needed. Using the above hypothetical implementation, this quantification could be achieved by using a sigma type:

Definition $\text{Signature_t } (t : T) : \text{Type} := \{s : \text{sig_index } S \mid \text{snd } (\text{sig } s) = t\}$.

This definition would be awkward to use since we would be obliged to handle equality proofs when talking about indices, i.e. terms of $\text{sig_index } S$, with a specific output type. We can in fact do better: while the *propositional* equality as used above would need our intervention, *definitional* equality — *conversion* — is handled by Coq. Hence we decide to implement a signature over a set of types T as a function that maps each $t : T$ to the collection of arities whose output type is the given t . In other words, the parameter t of Signature_t in the definition of signature replaces the second component of the arities:

7.2 Code (Signature, Def. 3.9):

Record $\text{Signature_t } (t : T) : \text{Type} := \{$
 $\quad \text{sig_index} : \text{Type} ;$
 $\quad \text{sig} : \text{sig_index} \rightarrow [[T] * T] \}$.

Definition $\text{Signature} := \text{forall } t, \text{Signature_t } t$.

We discuss the formalization of the example signature of the simply-typed lambda calculus (cf. Ex. 3.23). At first we define an indexing type TLC_index_t for each $t : T$. After that, we build an indexed signature TLC_sig mapping each index to its arity:

7.3 Code (Signature of TLC, Ex. 3.23):

Inductive $\text{TLC_index} : T \rightarrow \text{Type} :=$
 $\quad | \text{TLC_abs} : \text{forall } s \ t : T, \text{TLC_index } (s \sim> t)$
 $\quad | \text{TLC_app} : \text{forall } s \ t : T, \text{TLC_index } t$.

Definition $\text{TLC_arguments} : \text{forall } t, \text{TLC_index } t \rightarrow [[T] * T] :=$
 $\quad \text{fun } t' \ r \Rightarrow \text{match } r \text{ with}$
 $\quad \quad | \text{TLC_abs } s \ t \Rightarrow (s::\text{nil}, t)::\text{nil}$
 $\quad \quad | \text{TLC_app } s \ t \Rightarrow (\text{nil}, s \sim> t)::(\text{nil}, s)::\text{nil}$
 $\quad \text{end}$.

Definition $\text{TLC_sig } t := \text{Build_Signature_t } t \ (\text{@TLC_arguments } t)$.

7.1.2. Modules and Morphisms for Arities

To any signature given as a dependent function of type Signature as in Code 7.2 we associate the *actual* signature in the sense of Def. 3.18. More precisely, for an arity $s = ([(\mathbf{s}_1, t_1), \dots, (\mathbf{s}_n, t_n)], t_0)$ given by lists we define the functors $\text{dom}(s)$ and $\text{cod}(s)$, each of which, given a monad $P \in \text{Mon}(\text{Set}^T)$ (cf. Def. 3.13), yield a P -module with codomain Set . Note that the bold face letters \mathbf{s}_i denote lists of sorts.

It would in principle be possible to build the module $\text{dom}(s, P)$ associated to a monad P using the category-theoretic machinery defined in [Sects. 2.2.2](#) and [2.2.3](#), i.e. by applying iteratively the derivation functor to the tautological module P as often as indicated by the arity s and finally the suitable fibre functor, glueing everything together via the product on module categories. However, we choose not to, for reasons we explain now. Consider again the diagram of [Disp. \(3.2.3\)](#), instantiated for the classic arity s :

$$\begin{array}{ccc}
 \prod_{i=1}^n [P^{s_i}]_{t_i} & \xrightarrow{\alpha^P} & [P]_{t_0} \\
 \prod_i [f^{s_i}]_{t_i} \downarrow & & \downarrow f_{t_0} \\
 f^* \prod_{i=1}^n [Q^{s_i}]_{t_i} & \xrightarrow{f^*(\alpha^Q)} & f^*[Q]_{t_0}
 \end{array} \tag{7.1.1}$$

This diagram actually makes use of many instances of the equalities mentioned in [Rem. 2.63](#), in order to justify composability of module morphisms. For instance, in the lower right corner, the fact that pullback and fibre may be permuted, is used. In Coq the aforementioned equalities of modules hold propositionally (if one uses appropriate axioms, such as proof irrelevance), but not definitionally, i.e. the modules are not convertible (see also [Rem. 2.64](#)). In order to be able to compose a module morphism with target $\rho^*[M]_u$, for instance, with a module morphism with source module $[\rho^*M]_u$, one needs to insert a suitable isomorphism of modules $\rho^*[M]_u \cong [\rho^*M]_u$. The carriers of these isomorphisms are families of identity functions, respectively, since the carriers of the source and target modules are convertible. In our formalization we would have to insert these isomorphisms (called `PROD_PB`, `ITDER_PB` and `ITFIB_PB` in our Coq library) in order to make some compositions typecheck — as illustrated by the diagram in [Disp. \(7.1.2\)](#) — which in turn would result in quite a cumbersome formalization with decreased readability. Instead we decide to implement the left vertical morphism from scratch. For this to work it is most convenient to define the carrier of the product modules as an inductive type, instead of applying the product in the module category recursively. Hence also the product modules are built manually rather than using the categorical devices of derivation, fibre and product.

7.1.2.1. Domain, Codomain, Representations

Given an arity $s = (s_1, t_1), \dots, (s_n, t_n) \rightarrow t_0$ (or shorter $\ell \rightarrow t_0$) and a monad P , we have to construct the module

$$\text{dom}(s, P) = \prod_{i=1}^n [P^{s_i}]_{t_i} = \prod_{\ell} P .$$

7. Formalization of Zsidó's theorem

$$\begin{array}{ccc}
 \prod_{i=1}^n [P^{s_i}]_{t_i} & \xrightarrow{\alpha^P} & P_{t_0} \\
 \downarrow \Pi_i [f^{s_i}]_{t_i} & & \downarrow f_{t_0} \\
 \prod_{i=1}^n [(f^*Q)^{s_i}]_{t_i} & & \\
 \downarrow \Pi_i [\cong]_{t_i} & & \\
 \prod_{i=1}^n [f^*(Q^{s_i})]_{t_i} & & \\
 \downarrow \Pi_i \cong & & \\
 \prod_{i=1}^n f^*[Q^{s_i}]_{t_i} & & \\
 \downarrow \cong & & \\
 f^* \prod_{i=1}^n [Q^{s_i}]_{t_i} & \xrightarrow{f^*(\alpha^Q)} f^*[Q]_{t_0} \xrightarrow{\cong} [f^*Q]_{t_0} &
 \end{array} \tag{7.1.2}$$

Its carrier, being a kind of heterogeneous list, is given as an inductive type parametrized by a set family of variables and a list such as the list ℓ indicating the domain of an arity. In fact, for defining the carrier, only an object map M of the type indicated below is necessary:

Variable $M : (\text{ITYPE } T) \rightarrow (\text{ITYPE } T)$.

Inductive $\text{prod_mod_c } (V : \text{ITYPE } T) : [[T]^* T] \rightarrow \text{Type} :=$

| $\text{TTT} : \text{prod_mod_c } V \text{ nil}$

| $\text{CONSTR} : \text{forall } b \text{ bs,}$

$M (V ** (\text{fst } b)) (\text{snd } b) \rightarrow \text{prod_mod_c } V \text{ bs} \rightarrow \text{prod_mod_c } V (b::\text{bs}).$

Now, for a list $l : [[T]^* T]$, if M is equipped with a module structure over a monad P , we equip the map $\text{fun } V \Rightarrow \text{prod_mod_c } V \text{ l}$ with a module structure. Its substitution is given by a function pm_mkl , which is defined by recursion on the argument of type $\text{prod_mod_c } \dots$, applying the module substitution of M in each component:

Fixpoint $\text{pm_mkl } l \text{ V W } (f : V \dashrightarrow P \text{ W}) (X : \text{prod_mod_c } M \text{ V } l) :$
 $\text{prod_mod_c } M \text{ W } l :=$
 $\text{match } X \text{ in } \text{prod_mod_c } _ _ \text{ l return } \text{prod_mod_c } M \text{ W } l \text{ with}$
 | $\text{TTT} \Rightarrow \text{TTT } M \text{ W}$

```

| CONSTR b bs elem elems => CONSTR (M:=M) (V:=W)
    (mkleisli (Module_struct := M) (lshift f) (snd b) elem)
    (pm_mkl f elems)

end.

```

Proving its module property — by induction on the argument X — yields a module $\text{prod_mod } l$ for each list $l : [[T] * T]$. For $s = \ell \rightarrow t_0$, this defines the object function of the functor $\text{dom}(s)$. The object function of $\text{cod}(s)$ is easy to define, since it simply associates, to any monad P , the fibre module with respect to t_0 of the tautological module P . Again, this is defined more generally for any P -module M with codomain category Set^T . Putting both domain and codomain together, we associate, to any algebraic arity s and any P -module M , a type of module morphisms

$$\text{dom}(s, M) \rightarrow \text{cod}(s, M)$$

as in [Code 7.4](#) below. Note that M is later instantiated by the tautological P -module P .

7.4 Code (Representation of an Arity, [Def. 3.25](#)):

```

Variable M : Module P (ITYPE T).
Definition modhom_from_arity (ar : [[T] * T] * T) : Type :=
  Module_Hom (prod_mod M (fst ar)) (M [(snd ar)]).

```

where $M[(s)]$ denotes the fibre of the module M over s . Finally a representation of a signature S in a monad P is given by a module morphism for each arity i , i.e. by specifying a function of type

$$\forall s \in S, \text{dom}(s, P) \rightarrow \text{cod}(s, P) ,$$

where P denotes the tautological P -module. Since the set of arities is indexed by the target type of the arities, the representation structure is indexed as well:

7.5 Code (Representation of a Signature, [Def. 3.25](#)):

```

Variable P : Monad (ITYPE T).
Definition Repr_t (t : T) :=
  forall i : sig_index (S t), modhom_from_arity P ((sig i), t).
Definition Repr := forall t, Repr_t t.

```

We bundle the data and define a representation as a monad together with a representation structure over this monad¹:

```

Record Representation := {
  rep_monad :> Monad (ITYPE T);
  repr : Repr rep_monad }.

```

¹ Here an example of *coercion* occurs. The special notation $:>$ allows us to omit the projection rep_monad when accessing the monad which underlies a given representation R . We can hence also write $R \ x$ for the value of the monad of R on an object x of the underlying category.

7.1.2.2. Morphisms of Representations

The carrier of the domain module $\text{dom}(s, P) = \prod_l P$ of a representation (cf. [Disp. \(7.1.1\)](#)) is defined as an inductive type. This suggests the use of structural recursion for defining the left vertical morphism of the commutative diagram of [Disp. \(7.1.1\)](#). Given a monad morphism $f : P \rightarrow Q$, we apply f to every component of $\prod_l P$:

```
Fixpoint Prod_mor_c (l : [[T] * T]) (V : ITYPE T) (X : prod_mod P l V) :
  f* (prod_mod Q l) V :=
  match X in prod_mod_c _ _ l return f* (prod_mod Q l) V with
  | TTT => TTT _ _
  | CONSTR b bs elem elems =>
    CONSTR (f _ _ elem) (Prod_mor_c elems)
  end.
```

This function is easily proved to be a morphism of P -modules

$$\text{dom}(s, f) := \text{Prod_mor} : \prod_l P \rightarrow f^* \prod_l Q .$$

We thus are able to avoid mentioning all those trivial isomorphisms in the definition of the arrow map of the functor $\text{dom}(s)$ that are present in the diagram of [Disp. \(7.1.2\)](#).

The codomain arrow $\text{cod}(s, f) = f_{t_0}$ is obtained by taking the fibre module of the module morphism induced by f , cf. [Sect. 2.2.2](#). The Coq function `PbMod_ind_Hom`, which associates to any monad morphism the induced module morphism, can even be declared as a coercion

```
Coercion PbMod_ind_Hom : Monad_Hom >-> mor.
```

such that the abuse of notation introduced in the informal [Def. 2.52](#) has a counterpart in the formal development.

The isomorphism in the lower right corner however remains in the formalization, appearing as `ITPB_FIB`. Its underlying family of morphisms, however, is simply a family of identity functions. For an arity a and module morphisms `RepP` and `RepQ` representing this arity in monads P and Q respectively, the definition of the commutative diagram reads as follows:

7.6 Code (Commutative Diagram for Representation Morphism, [Def. 3.26](#)):

```
Definition commute f RepP RepQ : Prop :=
  RepP ;; f [(snd a)] == Prod_mor (fst a) ;; f* RepQ ;; ITPB_FIB f _ _
```

A morphism of representations from P to Q of the signature S is just a monad morphism from P to Q together with the commutativity property for each arity. More precisely, since arities are indexed by their target type, we have a commutative diagram for any object type $t : T$ and each arity (index) i in the indexing set of S t :

7.7 Code (Morphism of representations, [Def. 3.26](#)):

Variables $P\ Q : \text{Representation } S$.

Class `Representation_Hom_struct` ($f : \text{Monad_Hom } P\ Q$) :=
`repr_hom_s : forall t (i : sig_index (S t)), commute f (repr P i) (repr Q i).`

Record `Representation_Hom` : **Type** := {
`repr_hom_c :> Monad_Hom P Q;`
`repr_hom :> Representation_Hom_struct repr_hom_c }.`

As mentioned in [Sect. 3.2.2](#), representations of S and their morphisms form a category `REPRESENTATION S`. Composition of representations is defined by composing the underlying monad morphisms:

Program Instance `Rep_comp_struct` :
`Representation_Hom_struct (Monad_Hom_comp f g).`

where the commutation property is proved by some tactic defined beforehand. Accordingly, the identity morphism of representations is built upon the identity monad morphism:

Program Instance `Rep_Id_struct` :
`Representation_Hom_struct (Monad_Hom_id P).`

Since equality on morphisms of representations is defined as equality of the underlying monad morphisms, the properties of composition necessary for representations to form a category are a consequence of those for the category `MONAD (ITYPE T)`. The construction of the initial representation (and hence the proof of [Thm. 3.28](#)) is explained in the next section.

7.2. Construction of the Initial Object

The initial object of the category of representations of the signature S is constructed in several steps:

1. the syntax associated to S as an inductive data type `STS`,
2. definition of a monad structure `STS_Monad` on said data type,
3. construction of the representation structure `STSRepr` on `STS_Monad`,
4. for any representation R , construction of morphism `init R` from `STSRepr` to R ,
5. uniqueness of `init R` for any representation R .

7.2.1. The Terms Generated by a Signature

The first step is to define a map $\text{STS} : \text{ITYPE } T \dashrightarrow \text{ITYPE } T$ — the monad carrier — mapping each type family V of variables to the type family of terms with free variables

in V . Since objects of $\text{ITYPE } T$ really are dependent Coq types (cf. [Code 6.5](#)), this map is implemented as a Coq inductive family of types, parametrized by a context and dependent on object types. Apart from the use of dependent types, the “data” parts of this section could indeed be done in any programming language featuring inductive types.

Mutual induction is used, defining at the same time a type STS_list of heterogeneous lists of terms, yielding the arguments to the constructors of S . This list type is indexed by arities, such that the constructors can be fed with precisely the right kind of arguments.

7.8 Code (Terms of the Initial Representation):

```

Inductive STS (V : ITYPE T) : ITYPE T :=
  | Var : forall t, V t -> STS V t
  | Build : forall t (i : sig_index (S t)), STS_list V (sig i) -> STS V t
with
STS_list (V : ITYPE T) : [[T] * T] -> Type :=
  | TT : STS_list V nil
  | constr : forall b bs,
    STS (V * (fst b)) (snd b) -> STS_list V bs -> STS_list V (b::bs).

```

The constructor `Build` takes 3 arguments:

- an object type t indicating its output type,
- an arity i (resp. its index) from the set of indices with output type t and
- a term of type $\text{STS_list } V \text{ (sig } i)$ carrying the subterms of the term to construct.

Note that Coq typing ensures the correct typing of all constructible terms of STS , a technique called *intrinsic typing*. The `Scheme` command generates a mutual induction scheme for the defined pair of types. The latter type is actually isomorphic to the type prod_mod_c STS . This duplication of data could hence have been avoided by defining a nested inductive type as follows, instead of using mutual induction.

```

Inductive STS (V : ITYPE T) : ITYPE T :=
  | Var : forall t, V t -> STS V t
  | Build : forall t (i : sig_index (S t)), prod_mod_c STS V (sig i) -> STS V t.

```

However, we use the mutual inductive version because it allows us to define functions on those types by mutual recursion rather than by nested recursion; the latter are significantly more difficult to reason about.

7.2.2. Monad Structure on the Set of Terms

We continue by defining a monad structure on the map STS . Again, due to our choice of implementing sets as Coq types (cf. [Code 6.5](#)), the maps we need really are Coq functions.

As in the special case of ULC (cf. Ex. 2.36) and TLC (cf. Ex. 2.37), the monadic map η is given by the variable-as-term constructor `Var`. The substitution map `subst` is defined using two helper functions `rename` (providing functoriality, cf. Rem. 2.66) and `_shift` (used when substituting under binders, cf. Ex. 2.74). Renaming and substitution are implemented using mutual recursion on the mutually inductive data types `STS` and `STS_list`:

```

Fixpoint rename V W (f : V ----> W) t (v : STS V t) :=
  match v in STS _ t return STS W t with
  | Var t v => Var (f t v)
  | Build t i l => Build (i:=i) (list_rename l f)
  end
with
list_rename V t (l : STS_list V t) W (f : V ----> W) : STS_list W t :=
  match l in STS_list _ t return STS_list W t with
  | TT => TT W
  | constr b bs elem elems =>
      constr (elem // - ( f ^ (fst b)))
              (elems // -- f)
  end
where "x // - f" := (rename f x)
and "x // -- f" := (list_rename x f).
...
(* a lot more code *)
...
Fixpoint subst (V W : ITYPE T) (f : V ----> STS W) t (v : STS V t) :
  STS W t := match v in STS _ t return STS W t with
  | Var t v => f t v
  | Build t i l => Build (l >== f)
  end
with
list_subst V W t (l : STS_list V t) (f : V ----> STS W) : STS_list W t :=
  match l in STS_list _ t return STS_list W t with
  | TT => TT W
  | constr b bs elem elems =>
      constr (elem >== (_lshift f)) (elems >== f)
  end
where "x >== f" := (subst f x)
and "x >== f" := (list_subst x f).

```

The monadic properties that the substitution should satisfy, are similar to the lemmas one would prove in order to establish “programm correctness”. As an example, the third

monad law reads as

Lemma subst_subst V t (v : STS V t) W X (f : V ----> STS W)
 (g : W ----> STS X) :
 v >== f >== g = v >== f;; subst g.

Proof.

apply (@STSind
 (fun (V : T -> Type) (t : T) (v : STS V t) => forall (W X : T -> Type)
 (f : V ----> STS W) (g : W ----> STS X),
 v >== f >== g = v >== (f;; subst g))
 (fun (V : T -> Type) l (v : STS_list V l) =>
 forall (W X : T -> Type)
 (f : V ----> STS W) (g : W ----> STS X),
 v >>== f >>== g = v >>== (f;; subst g)));
 t5.

Qed.

Its proof script is a typical example; most of those lemmas are proved using the induction scheme STSind — instantiated with suitable properties — followed by a single custom tactic which finishes off the resulting subgoals, mainly by rewriting with equalities proved beforehand. After a quite lengthy series of lemmas we obtain that the function subst and the variable-as-term constructor Var turn STS into a monad:

Program Instance STS_monad : Monad_struct STS := {
 weta := Var ;
 kleisli := subst }.

7.2.3. A Representation in the Monad of Terms

The representational structure on STS is defined using the Build constructor. For each arity i in the index set sig_index (S t), we must give a morphism of modules from prod_mod STS (sig i) to STS [(t)]. Since the constructor Build takes its argument from STS_list and not from the isomorphic prod_mod STS, we precompose with one of the isomorphisms between those two types:

Program Instance STS_arity_rep (t : T) (i : sig_index (S t)) :
 Module_Hom_struct
 (S := prod_mod STS (sig i)) (T := STS [(t)])
 (fun V X => Build (STS_l_f_pm X)).

The only property to verify is the compatibility of this map with the module substitution, which we happily leave to Coq. We obtain a representation of S:

Record STSRepr : REPRESENTATION S := Build_Representation (@STSrepr).

7.2.4. Weak Initiality for the Representation in the Term Monad

In the introduction, we gave the equations that a morphism of representations of the natural numbers should satisfy. Reading those equations as a rewrite system from left to right yields a way to define iterative functions on the natural numbers. This idea is also used in order to define a morphism from STSRepr to any representation R of the signature S : a term of STS, whose root is a constructor $\text{Build } t \ i$ for some object type t and an arity i , is mapped recursively to the image — of the recursively computed argument — under the corresponding representation $\text{repr } R \ i$ of R . This definition for a morphism of representations will turn out to be the only one possible, leading to uniqueness. Formally, the carrier init of what will be the initial morphism from STSRepr to R is defined as a mutually recursive Coq function:

```

Fixpoint init V t (v : STS V t) : R V t :=
  match v in STS _ t return R V t with
  | Var t v => weta (Monad_struct := R) V t v
  | Build t i X => repr R i V (init_list X)
  end
with
init_list l (V : ITYPE T) (s : STS_list V l) : prod_mod R l V :=
  match s in STS_list _ l return prod_mod R l V with
  | TT => TTT _ _
  | constr b bs elem elems =>
    CONSTR (init elem) (init_list elems)
  end.

```

where the function init_list applies init to (heterogeneous) lists of arguments. We have to show that this function is a morphism of monads and a morphism of representations. A series of lemmas show that init commutes with renaming resp. lifting (init_lift), shifting (init_shift) and substitution (init_kleisli):

Lemma $\text{init_lift } V \ t \ x \ W \ (f : V \dashrightarrow W) : \text{init } (x // - f) = \text{lift } f \ t \ (\text{init } x).$
Lemma $\text{init_shift } a \ V \ W \ (f : V \dashrightarrow \text{STS } W) : \text{forall } (t : T) \ (x : \text{opt } a \ V \ t),$
 $\text{init } (x >> - f) = x >> - (f ;; @\text{init } _).$
Lemma $\text{init_kleisli } V \ t \ (v : \text{STS } V \ t) \ W \ (f : V \dashrightarrow \text{STS } W) :$
 $\text{init } (v >== f) = \text{kleisli } (f ;; @\text{init } _) \ t \ (\text{init } v).$

The latter property is precisely one of the axioms of morphisms of monads (cf. [Def. 3.12](#), rectangular diagram). The second monad morphism axiom which states compatibility with the η s of the monads involved is fulfilled by definition of init — it is exactly the first branch of the pattern matching by which the function init is defined. We hence have established that init is (the carrier of) a morphism of monads:

Program Instance `init_monadic` : `Monad_Hom_struct` (P:=STSM) `init`.

Record `init_mon` := `Build_Monad_Hom` `init_monadic`.

Very much less work is then needed to show that `init` also is a morphism of representations:

Program Instance `init_representic` : `Representation_Hom_struct` `init_mon`.

7.2.5. Uniqueness and Initiality

Uniqueness of the morphism of representations `init_rep` (obtained from packaging `init_representic` into a record instance) is expressed by the following lemma:

Lemma `init_unique` : `forall` `f` : `STSRepr` \longrightarrow `R` , `f` == `init_rep`.

Instead of directly proving the lemma, we prove at first an unfolded version which allows to directly apply the mutual induction scheme `STSind`:

Variable `f` : `Representation_Hom` `STSRepr` `R`.

Hint Rewrite `one_way` : `fin`.

Ltac `t1` := `tt`;

```
(try match goal with [t:T, s : STS_list _ _ | _] => rewrite <- (one_way s);
  let H:=fresh in assert (H:=repr_hom f (t:=t));
  unfold commute in H; simpl in H end);
repeat (app (mh_weta f) || tinv || tt).
```

Lemma `init_unique_prepa` `V` `t` (`v` : `STS` `V` `t`) : `f` `V` `t` `v` = `init` `v`.

Proof.

```
apply (@STSind
  (fun V t v => f V t v = init v)
  (fun V l v => Prod_mor f l V (pm_f_STSl v) = init_list v));
ttt.
```

Qed.

Finally we declare an instance of the `Initial` type class for the category of representations `REPRESENTATION` `S` with `STSRepr` as initial object and `init_rep` `R` as the initial morphism towards any other representation `R`.

7.9 Code (Instance of `Initial` for Category of Representations):

Program Instance `STS_initial` : `Initial` (`REPRESENTATION` `S`) := {
`Init` := `STSRepr` ;
`InitMor` `R` := `init_rep` `R` }.

In this instance declaration, the proof field `InitMorUnique` is filled automatically by the **Program** feature, using the preceding lemma `init_unique`.

7.3. Remarks

The nature of the theorem made it convenient for computer theorem proving: the proofs are straightforward, carrying no surprises. Moreover, they are highly technical using (mutual) induction, something Coq offers good support for.

Some aspects remain unsatisfactory: using type classes and records simultaneously is at least confusing for the reader, even if there are reasons from the implementor's point of view to do so. Also, the weak support for nested induction in Coq obliged us to use mutual induction instead, leading to some duplication of data and hence another unnecessary source of confusion. Other aspects, such as the implementation of syntax in an efficient way, i.e. without any extrinsic typing device, could be done due to Coq's good support for dependent types.

According to `coqwc`² the Coq files that are specific to the proved theorem consist of approximately 400 lines of specification and 600 lines of proof. The proofs are done in a semi-automated way, employing a proof style promoted by Chlipala in his online book [Ch1], as well as in a published user tutorial [Ch10]. An earlier version using a more standard proof style included about 900 lines of proof. This reduction is mainly due to the fact that proof automation also stimulates reuse of code – here reuse of proof code – similarly to how polymorphism does for data structures and functions. However, we do not claim to be experts in proof automation, nor do we have “one tactic to rule them all”.

²The tool `coqwc`, part of the standard Coq tools, counts the number of lines in a Coq source file, classified into the 3 categories *specification*, *proof* and *comment*.

8. INITIALITY FOR UNTYPED 2–SIGNATURES, FORMALIZED

In this chapter we present the formalization in the proof assistant Coq of [Thm. 4.34](#) of [Chapt. 4](#). We first define arities and 1–signatures in terms of lists. Afterwards we define representations for 1–arities and construct the initial such representation. We then formalize inequations over 1–signatures and construct, for any suitable 2–signature, the initial representation. Finally we show how to specify the untyped lambda calculus with beta reduction via a 2–signature.

8.1. Arities by Lists

According to [Def. 4.1](#), a 1–signature consists of an indexing type and, for each index, a list of natural numbers, indicating the number of arguments of a constructor, as well as the number of variables bound in each argument. Formally, 1–signatures are an untyped version of [Code 7.1](#). In the formalization they are simply called “signatures”:

8.1 Code (1–Signature, [Def. 4.1](#)):

```

Notation "[ T ]" := (list T) (at level 5).
Record Signature : Type := {
  sig_index : Type ;
  sig : sig_index -> [nat] }.

```

Next we formalize context extension according to a natural number, cf. [Sect. 2.4.3](#). These definitions are important for the definition of the module morphisms we associate to an arity, cf. below. Context extension is actually functorial. Given a natural number n and a set of variables V , we recursively define the set V^{**n} to be the set V enriched with n additional variables.

8.2 Code (Adding fresh variables):

```

Fixpoint pow (n : nat) (V : TYPE) : TYPE :=
  match n with
  | 0 => V
  | S n' => pow n' (option V)
  end.

```

Notation " V^{**n} " := (pow n V) (at level 10).

Fixpoint pow_map (l : nat) V W (f : V → W) :
 $V^{**l} \rightarrow W^{**l} :=$
 match l return $V^{**l} \rightarrow W^{**l}$ with
 | 0 => f
 | S n' => pow_map (^ f)
 end.

Notation " f^{ll} " := (pow_map (l:=l) f) (at level 10).

8.2. Representations of a 1-Signature

Given a classic arity s , i.e. a list of natural numbers s (cf. [Code 8.1](#)), and a relative monad P on the functor Δ , we define the product module P^s as in [Rem. 4.9](#). More generally, we define M^s for any P -module M with codomain Pre . Analogously to the implementation of [Chapt. 7](#), we build this module from scratch instead of relying on the category-theoretic constructions such as product and derivation functor for the module categories, allowing us to omit the insertion of isomorphisms in the style of [Lem. 2.106](#) and [2.107](#). Given any module M over a monad P from sets to preordered sets, we define the product type `prod_mod_c` as a dependent type parametrized by a set of variables and dependent on a list of naturals. Actually we define at first the carrier depending not on a module, but just on a carrier function M . The relation on the product is induced by that on M .

8.3 Code (Product Module, Carrier map):

Variable M : TYPE → Ord.

Inductive prod_mod_c (V : TYPE) : [nat] → Type :=

| TTT : prod_mod_c V nil
 | CONSTR : forall b bs,
 M (V ** b) → prod_mod_c V bs → prod_mod_c V (b::bs) .

Notation " $a \text{--}:-\text{ } b$ " := (CONSTR a b) (at level 60).

Inductive prod_mod_c_rel (V : TYPE) : forall n, relation (prod_mod_c M V n) :=

| TTT_rel : forall x y : prod_mod_c M V nil, prod_mod_c_rel x y
 | CONSTR_rel : forall n l, forall x y : M (V ** n),
 forall a b : prod_mod_c M V l, x << y →
 prod_mod_c_rel a b → prod_mod_c_rel (x --:- a) (y --:- b).

Note that the infix " $<<$ " is overloaded notation and denotes the relation of any preordered set. For any given list a of naturals and any set V of variables, the set `prod_mod_c V a` equipped with the relation `prod_mod_c_rel V a` is in fact a preordered set. For the proof of transitivity we rely on the Coq tactic [dependent induction](#),

thus on the axioms

```
JMeq.JMeq_eq : forall (A : Type) (x y : A), x ~ y -> x = y
Eqdep.Eq_rect_eq.eq_rect_eq : forall (U : Type) (p : U)
  (Q : U -> Type) (x : Q p)
  (h : p = p), x = eq_rect p Q x p h
```

from the Coq standard library.

Now, if M is not just a map of type $\text{TYPE} \rightarrow \text{Ord}$, but a module over some relative monad P over Δ , we equip the product map with a modhic substitution in form of a recursive function:

8.4 Code (Product module, substitution):

Variable $M : \text{RMOD } P \Delta$.

```
Fixpoint pm_mkl l V W (f : Delta V ----> P W)
  (X : prod_mod_c (fun V => M V) V l) : prod_mod_c _ W l :=
  match X in prod_mod_c _ _ l return prod_mod_c (fun V => M V) W l
  with
  | TTT => TTT _ W
  | elem -:- elems =>
    rmkleisli (RModule_struct := M) (lshift _ f) elem -:- pm_mkl f elems
end.
(* ... *)
```

```
Definition prod_mod (a : [nat]) := Build_RModule (prod_mod_struct a).
```

Afterwards we prove by induction that this map is indeed monotone with respect to the preorder defined in [Code 8.3](#). Altogether, [Code 8.3](#) and [8.4](#) define a module $\text{prod_mod } M \ l$ for any module $M : \text{RMOD } P \text{ Ord}$ and any list of naturals l .

To any arity $\text{ar} : [\text{nat}]$ and a module M over a monad P we associate a type of module morphisms $\text{modhom_from_arity } \text{ar } M$. Representing ar in M then means giving a term of type $\text{modhom_from_arity } \text{ar } M$. Note that in the corresponding [Def. 4.13](#) we have defined representations in *monads* only. Indeed we instantiate M with the tautological module later.

8.5 Code (Type of Representations of an Arity, [Def. 4.13](#)):

Variable $P : \text{RM Monad } \Delta$.

```
Definition modhom_from_arity (M : RModule P Ord) (ar : [nat]) : Type :=
  RModule_Hom (prod_mod M ar) M.
```

For the rest of the section, we suppose a signature S to be given via a Coq section variable, **Variable** $S : \text{Signature}$. As just mentioned, representing the signature S in a monad P (cf. [Def. 4.14](#)) means providing a suitable module morphism for any arity

of S , i.e. providing, for any element of the indexing set $\text{sig_index } S$, a term of type $\text{modhom_from_arity } P \text{ (sig } i)$:

8.6 Code (Representation of 1-Signature, [Def. 4.14](#)):

```

Definition Repr (P : RMonad Delta) :=
  forall i : sig_index S, modhom_from_arity P (sig i).
Record Representation := {
  rep_monad :> RMonad Delta ;
  repr : Repr rep_monad }.

```

The projection rep_monad is declared as a *coercion* by using the special syntax $:>$. This coercion allows for abuse of notation in Coq as we do informally according to [Def. 4.14](#). See the first paragraph of [Sect. 8.6](#) for a use of this abuse.

8.3. Morphisms of Representations

A morphism of representations from P to Q is given by a monad morphism $f : P \rightarrow Q$ between the underlying monads such that a diagram commutes for any arity, cf. [Def. 4.17](#). The main task in the implementation is to define this diagram for a given arity ℓ , and, more specifically, the left vertical morphism

$$\text{dom}(\ell, f) = f^\ell : P^\ell \rightarrow f^* Q^\ell .$$

using the notation of [Rem. 4.9](#). Since P^ℓ is defined as an inductive type, it makes sense to define f^ℓ by recursion on the inductive type underlying P^ℓ , named $\text{prod_mod_c } P \text{ V } l$ (cf. [Code 8.3](#)):

8.7 Code (Carrier of Domain Module Morphism of [Def. 4.17](#)):

```

Variables P Q : RMonad Delta.
Variable f : RMonad_Hom P Q.
Fixpoint Prod_mor_c (l : [nat]) (V : TYPE) (X : prod_mod_c (fun V => P V) V
  l) :
  (prod_mod_c _ V l) :=
  match X in prod_mod_c _ _ l
  return f* (prod_mod Q l) V with
  | TTT => TTT _ _
  | elem _ :- elems => f _ elem _ :- Prod_mor_c elems
  end.

```

Proving this map monotone is a simple exercise, as well as its commutation property with substitution, yielding the aforementioned module morphism. Now we have all

the ingredients we need in order to define the diagram of [Def. 4.17](#). For an arity a the diagram reads as follows:

8.8 Code (Commutative Diagram of [Def. 4.17](#)):

```
Variable a : [nat].
Variable RepP : modhom_from_arity P a.
Variable RepQ : modhom_from_arity Q a.
Notation "f * M" := (# (PbRMOD f _ ) M).
Definition commute := Prod_mor a ;; f * RepQ == RepP ;; f^.
```

Here $f^$ denotes the module morphism induced by a monad morphism, cf. [Def. 2.100](#). Using the preceding definition, we define morphisms of representations of S :

8.9 Code (Morphism of Representations, [Def. 4.17](#)):

```
Variables P Q : Representation.
Class Representation_Hom_struct (f : RMonad_Hom P Q) :=
  repr_hom_s : forall i : sig_index S,
    commute f (repr P i) (repr Q i).
Record Representation_Hom : Type := {
  repr_hom_c :> RMonad_Hom P Q;
  repr_hom :> Representation_Hom_struct repr_hom_c }.
```

8.4. Category of Representations

In this section we describe in more detail the category of representations of a 1–signature, cf. [Def. 4.19](#). The composition of morphisms of representations $f : P \rightarrow Q$ and $g : Q \rightarrow R$ is essentially done by composing the underlying monad morphisms. One has to show that this morphism does indeed commute with the representation morphisms of P and R . Similarly, the identity monad morphism of (the monad underlying) a representation P yields a morphism of representations. Fed with some suitable lemma, the [Program](#) framework does the job for us:

8.10 Code (Composition and Identity of Representations):

```
Variables P Q R : Representation S.
Variable f : Representation_Hom P Q.
Variable g : Representation_Hom Q R.
Program Instance Rep_comp_struct :
  Representation_Hom_struct (RMonad_comp f g).
Program Instance Rep_Id_struct : Representation_Hom_struct (RMonad_id P).
```

Since equality of morphisms of representations is defined as equality of the underlying monad morphisms, the categorical properties of composition are established already as part of the definition of the category $\text{RMONAD } F$ for any functor F .

8.11 Code (Category of Representations, [Def. 4.19](#)):

```
Program Instance REP_struct : Cat_struct (@Representation_Hom S) := {
  mor_oid a c := eq_Rep_oid a c;
  id a := Rep_Id a;
  comp P Q R f g := Rep_Comp f g }.
```

Definition $\text{REP} := \text{Build_Cat REP_struct}$.

8.5. Initiality without Inequations

We construct the initial object of the category REP (cf. [Code 8.11](#)). In the informal proof of [Lem. 4.21](#) this initial object is the image under a left adjoint of the initial object in a category of representations as defined in [Sect. 3.2](#) with the set of object sorts $T = \{*\}$. For the formal proof we decide to implement the initial object of REP directly, in order to obtain a compact formalization. However, the initial object is constructed in a way similar to that of [Chapt. 7](#). The carrier of the initial representation is just a simplified — because untyped — version of [Code 7.8](#). The only significant difference to [Chapt. 7](#) is that we equip the set of terms with the trivial diagonal preorder by applying the functor Δ , in Coq called Delta :

8.12 Code:

```
Inductive UTS (V : TYPE) : TYPE :=
  | Var : V -> UTS V
  | Build : forall (i : sig_index S), UTS_list V (sig i) -> UTS V
with
UTS_list (V : TYPE) : [nat] -> Type :=
  | TT : UTS_list V nil
  | constr : forall b bs,
    UTS (V ** b) -> UTS_list V bs -> UTS_list V (b::bs).
```

Notation $"a \dashv\vdash b" := (\text{constr } a \ b)$.

Definition $\text{UTS_sm } V := \text{Delta (UTS } V)$.

We define renaming and, built on top of renaming, substitution:

```
Fixpoint rename (V W: TYPE) (f : V -> W) (v : UTS V) :=
  match v in UTS _ return UTS W with
  | Var v => Var (f v)
  | Build i l => Build (i // f)
```

```

    end
  with
    list_rename V t (l : UTS_list V t) W (f : V ----> W) : UTS_list W t :=
      match l in UTS_list _ t return UTS_list W t with
      | TT => TT W
      | constr b bs elem elems => elem // - f ^ b -:: - elems // - f
      end
  where "x // - f" := (rename f x)
  and "x // - f" := (list_rename x f).
  Fixpoint subst (V W : TYPE) (f : V ----> UTS W) (v : UTS V) :
    UTS W := match v in UTS _ return UTS _ with
    | Var v => f v
    | Build i l => Build (l >>= f)
    end
  with
    list_subst V W t (l : UTS_list V t) (f : V ----> UTS W) : UTS_list W t :=
      match l in UTS_list _ t return UTS_list W t with
      | TT => TT W
      | elem -:: - elems =>
        elem >= _ lshift f -:: - elems >>= f
      end
  where "x >= f" := (subst f x)
  and "x >>= f" := (list_subst x f).

```

Accordingly, the definition of a monadic structure on $V \mapsto \Delta \text{UTS}(V)$ differs from the monad STS_monad of Sect. 7.2 only in the occasional use of the functor Δ (Delta) on the morphisms — corresponding to the definition of the left adjoint for Lem. 4.5:

8.13 Code (Relative Monad Freely Generated by 1–Signature):

Program Instance $\text{UTS_sm_rmonad} : \text{RMonad_struct } \Delta \text{ UTS_sm} := \{$
 $\text{rweta } c := \# \Delta (\text{@Var } c);$
 $\text{rkleisli } a \ b \ f := \# \Delta (\text{subst } f) \}.$

Canonical Structure $\text{UTSM} := \text{Build_RMonad } \text{UTS_sm_rmonad}.$

The monad UTSM is easily equipped with a representation of the signature S ; the carrier of the representation of $i : \text{sig_index } S$ is given by the function

$\text{fun } (X : \text{prod_mod_c } _ \ V \ (\text{sig } i)) \Rightarrow \text{Build } (i := i) \ (\text{UTSI_f_pm } (V := V) \ X)$

that is, by the constructor $\text{Build } i$ of the type UTS , precomposed with an isomorphism UTSI_f_pm from $\text{prod_mod_c } \text{UTS}$ to UTS_list . We thus obtain a representation UTSRepr of the signature S .

Given another representation, say, R , of S , the morphism init from UTSRepr to R is defined by recursion:

```

Fixpoint init V (v : UTS V) : R V :=
  match v in UTS _ return R V with
  | Var v => rweta (RMonad_struct := R) V v
  | Build i X => repr R i V (init_list X)
  end
with
  init_list l (V : TYPE) (s : UTS_list V l) : prod_mod R l V :=
  match s in UTS_list _ l return prod_mod R l V with
  | TT => TTT _ _
  | elem -::: elems => init elem -::: init_list elems
  end.

```

This map `init` is compatible with lifting and substitution in `UTSM` and `R`, respectively:

```

Lemma init_lift V x W (f : V ----> W) :
  init (x / / - f) = rlift R f (init x).
Lemma init_kleisli V (v : UTS V) W (f : Delta V ----> UTS_sm W) :
  init (v >== f) = rkleisli (f ;; @init_sm W) (init v).

```

where `init_sm W` is the (trivially) monotone version of `init W` — the adjunct of `init W` under the adjunction of [Lem. 2.18](#). The latter of those lemmas constitutes an important part of the proof that `init` is the carrier of a module morphism from `UTSM` to `R`. It is trivial to prove that `init` is also compatible with the representation structure of `UTSRepr` and `R`, thus the carrier of a morphism of representations called `init_rep : UTSRepr ----> R`. Afterwards uniqueness of `init_rep` is proved:

```

Lemma init_unique : forall f : UTSRepr ----> R , f == init_rep.

```

Finally we establish initiality by an instance declaration of the corresponding class:

```

Program Instance UTS_initial : Initial (REP S) := {
  Init := UTSRepr ;
  InitMor R := init_rep R }.

```

8.6. Inequations and Initial Representation of a 2-Signature

For a 1-signature S , an S -module is defined to be a functor from representations of S to the category whose objects are pairs of a monad P and a module M over P , cf. [Def. 4.22](#). We do not need the functor properties, and use dependent types instead of the cumbersome category of pairs, in order to ensure that a representation in a monad P is mapped to a P -module.

The below definition makes use of two *coercions*. Firstly, we may write $a : \mathcal{C}$ because the “object” projection of the category record (cf. [Code 6.3](#)) is declared as a coercion. Secondly, the monad underlying any representation can be accessed without explicit projection using the coercion in [Code 8.6](#) we mentioned above.

```
Record S_Module := {
  s_mod :> forall R : REP S, RMOD R wOrd ;
  s_mod_hom :> forall (R T : REP S)(f : R ----> T),
    s_mod R ----> PbRMod f (s_mod T) }.
Notation "U @ f" := (s_mod_hom U f)(at level 4).
```

Note that we write $U@f$ for the image of the morphism of representations f under the S -module U . Source and target module of f are implicit arguments in this application.

A half-equation is a natural transformation between S -modules. We need the naturality condition in the following. Since we have not formalized S -modules as functors, we have to state naturality explicitly:

8.14 Code (Half-Equation, [Def. 4.22](#)):

```
Class half_equation_struct (U V : S_Module)
  (half_eq : forall R : REP S, U R ----> V R) := {
  comm_eq_s : forall (R T : REP S) (f : R ----> T),
    U @ f ;; PbRMod_Hom _ (half_eq T) == half_eq R ;; V @ f }.
Record half_equation (U V : S_Module) := {
  half_eq :> forall R : REP S, U R ----> V R ;
  half_eq_s :> half_equation_struct half_eq }.
```

We now formalize *classic* S -modules. Any list of natural numbers uniquely specifies a classic S -module, cf. [Def. 4.26](#). Given a list of naturals codl , we call this S -module $S_Mod_classic\ \text{codl}$. A *classic half-equation* is any half-equation with a classic codomain, and a classic inequation is a pair of parallel classic half-equations (cf. [Def. 4.33](#)):

```
Definition half_eq_classic (U : S_Module)(codl : [nat]) :=
  half_equation U (S_Mod_classic codl).
```

```
Record ineq_classic := {
  Dom : S_Module ;
  Cod : [nat] ;
  eq1 : half_eq_classic Dom Cod ;
  eq2 : half_eq_classic Dom Cod }.
```

Given a representation P and a (classic) inequation e , we check whether P satisfies e by pointwise comparison (cf. [Def. 4.31](#)):

```
Definition satisfies_ineq (e : ineq_classic) (P : REP S) :=
  forall c (x : Dom e P c),
```

8. Initiality for Untyped 2-Signatures, Formalized

```

    eq1 _ _ _ x << eq2 _ _ _ x.
(* for a family of inequations indexed by a set A *)
Definition Inequations (A : Type) := A -> ineq_classic.
Definition satisfies_ineqs A (T : Inequations A) (R : REP S) :=
  forall a, satisfies_ineq (T a) R.

```

We formalize sets of classic inequations as pairs of an indexing type A together with a term of type `Inequations A`, that is, a map from A to the type of classic inequations `ineq_classic`. The category of representations of (S, A) is obtained as a full subcategory of the category of representations of S . The following declaration produces a subcategory from predicates on the type of representations and on the (dependent) type of morphisms of representations, yielding the category `PROP_REP` of representations of (S, A) :

```

Variable A : Type.
Variable T : Inequations A.
Program Instance Ineq_Rep : SubCat_compat (REP S)
  (fun P => satisfies_ineqs T P) (fun a b f => True).
Definition INEQ_REP : Cat := SubCat Ineq_Rep.

```

We now construct the initial object of `INEQ_REP`. The relation on the initial object is defined precisely as in the paper proof, cf. [Disp. \(4.4.1\)](#):

```

Definition prop_rel_c X (x y : UTS S X) : Prop :=
  forall R : PROP_REP, init (FINJ _ R) x << init (FINJ _ R) y.

```

Here, `FINJ _ R` denotes the representation R as a representation of S , i.e. the injection of R in the category `REP S` of representations of S . The relation defined above is indeed a preorder, and we define the monad `UTSP` to be the monad whose underlying sets are identical to `UTSM`, namely the sets defined by `UTS`, but equipped with this new preorder. This monad `UTSP` is denoted by Σ_A in the paper proof.

The representation module morphisms of the initial representation `UTSRepr` can be “reused” after having proved their compatibility with the new order, yielding a representation `UTSPProp`. An important lemma states that this representation satisfies the inequations of T :

```

Lemma UTSPRepr_sig_prop : satisfies_ineqs T UTSPProp.

```

We have to explicitly inject the representation into the category of representations of (S, A) :

```

Definition UTSPROP : INEQ_REP :=
  exist (fun R : Representation S => satisfies_ineqs T R) UTSPProp
  UTSPRepr_sig_prop.

```

For building the initial morphism towards any representation $R : \text{INEQ_REP}$, we first build the corresponding morphism in the category of representations of S :

Definition `init_prop_re : UTSPopr ----> (FINJ _ R) := ...`

which we then inject, analogously to the initial representation, into the subcategory of representations of (S, A) :

Definition `init_prop : UTSPROP ----> R := exist _ (init_prop_re R) l.`

Finally we prove [Thm. 4.34](#): An initial object of a category is given by an object `Init` of this category, a map associating to any object `R` a morphism `InitMor R : Init ----> R`, and a proof of uniqueness of any such morphism. We instantiate the type class `Initial` for the category `INEQ_REP` of representations of (S, A) :

Program Instance `INITIAL_INEQ_REP : Initial INEQ_REP := {`
`Init := UTSPROP ;`
`InitMor := init_prop ;`
`InitMorUnique := init_prop_unique }.`

We check its type after closing all the sections — and thus abstracting from the section variables:

Check `INITIAL_INEQ_REP.`
`INITIAL_INEQ_REP`
`: forall (S : Signature) (A : Type) (T : Inequations S A),`
`Initial (INEQ_REP (S:=S) (A:=A) T)`

8.7. $\Lambda\beta$: Lambda Calculus with beta reduction

We implement the example 2–signature $\Lambda\beta$, cf. [Ex. 4.38](#). Throughout this section, we use a custom notation in Coq for the datatype of lists:

Notation `"[[x ; .. ; y]]" := (cons x .. (cons y nil) ..).`

In order to specify the 1–signature Λ (cf. [Def. 4.11](#), [Ex. 4.2](#)), we first define an indexing set `Lambda_index` consisting of two elements, `ABS` and `APP`. This indexing set reflects the fact that the signature Λ consists of two arities. The record instance `Lambda` is a term of type `Signature` (cf. [Code 8.1](#)). The map `sig` `Lambda` then associates the corresponding lists of naturals to each of these elements, according to [Ex. 4.2](#):

Inductive `Lambda_index := ABS | APP.`

Definition `Lambda : Signature := {`
`sig_index := Lambda_index ;`
`sig := fun x => match x with`
`| ABS => [[1]]`
`| APP => [[0 ; 0]]`
`end }.`

The definition of the inequation β (cf. [Ex. 4.32](#)) is a more challenging task, since a half-equation is not just an element of a simple datatype like a 1-arity, but given by suitable module morphisms.

At first, we define the substitution of *one* variable (cf. [Def. 4.27](#)) as a half-equation. The carrier `subst_carrier` of the substitution is defined as in [Def. 2.110](#). Afterwards we prove that this carrier satisfies the properties of a module morphism, that is, is compatible with substitution in the source and target modules. After abstracting from the section variable `R`, we obtain a function `subst_module_mor` which, given any representation `R` of `S`, yields the substitution module morphism associated to (the monad underlying) `R`.

Variable `S` : Signature.

Variable `R` : REP `S`.

Definition `subst_carrier` :

```
(forall c : TYPE, (S_Mod_classic_ob [[1; 0]] R) c ---->
  (S_Mod_classic_ob [[0]] R) c) := ...
```

Program Instance `sub_struct` : RModule_Hom_struct

```
(M:=S_Mod_classic_ob [[1; 0]] R)
```

```
(N:=S_Mod_classic_ob [[0]] R)
```

```
subst_carrier.
```

Definition `subst_module_mor` := Build_RModule_Hom (sub_struct `R`).

The last step is to prove “naturality”, that is, the commutativity of the family of diagrams of [Code 8.14](#). We recall that we do not implement *S*-modules as functors, but just as the data part of functors. This is why we put the word *naturality* in quotes. After the proof we define our first half-equation, `subst_half_eq`.

Program Instance `subst_half_s` : half_equation_struct

```
(U:= S_Mod_classic [[1; 0]])
```

```
(V:= S_Mod_classic [[0]])
```

```
subst_module_mor.
```

Definition `subst_half_eq` := Build_half_equation `subst_half_s`.

The definition of the second half-equation of [Ex. 4.28](#) is possible for any 1-signature with abstraction and application, such as the 1-signature Λ . To keep the example simple, we only define the half-equation for Λ . The needed steps are precisely the same as for the substitution half-equation, so we just give the statements.

Definition `beta_carrier` :

```
(forall c : TYPE, (S_Mod_classic_ob [[1; 0]] R) c ---->
  (S_Mod_classic_ob [[0]] R) c) := ...
```

Program Instance `beta_struct` : RModule_Hom_struct

```
(M:=S_Mod_classic_ob [[1; 0]] R)
```

```
(N:=S_Mod_classic_ob [[0]] R)
```

```
beta_carrier.
```

Definition `beta_module_mor` := `Build_RModule_Hom beta_struct`.

Program Instance `beta_half_s` : `half_equation_struct`

(`U`:=`S_Mod_classic Lambda [[1 ; 0]]`)

(`V`:=`S_Mod_classic Lambda [[0]]`)

`beta_module_mor`.

Definition `beta_half_eq` := `Build_half_equation beta_half_s`.

In the end we package both half-equations into one inequation specifying the beta rule of Ex. 4.32.

Definition `beta_rule` : `ineq_classic Lambda` := { |

`eq1` := `beta_half_eq` ;

`eq2` := `subst_half_eq Lambda` | }.

We can now associate a short name to the category of representations of $\Lambda\beta$, where, for increased clarity, we specify the implicit arguments:

Definition `Lambda_beta_Cat` := `INEQ_REP`

(`S`:=`Lambda`)(`A`:=`unit`)(`fun` `x` : `unit` => `beta_rule`).

Note that our formal definition allows that an inequation appears multiple times in a 2-signature, whereas in the informal definition we have *sets* of inequations. Unlike for arities, having several copies of the same inequation does not change the resulting category neither the initial object, of course. The initial representation is obtained via the specification

Definition `Lambda_beta` := `@Init`

(`INITIAL_INEQ_REP` (`fun` `x` : `unit` => `beta_rule`)).

9. A FAITHFUL TRANSLATION OF PCF TO ULC

In this chapter we describe the implementation of the category of representations of PCF, equipped with reduction rules — we refer to it as *semantic* PCF from now on — as described informally in [Sect. A.2](#). We state the reduction rules more precisely later. This theorem is an instance of [Thm. 5.21](#) proved in [Chapt. 5](#). However, for the implementation in Coq of this instance we make several simplifications compared to the general theorem:

- we do not define a notion of 2–signature, but specify directly a Coq type of representations of semantic PCF;
- we use dependent Coq types to formalize arities of higher degree (cf. [Def. 5.3](#)), instead of relying on modules on categories with pointed index sets. A representation of an arity of degree n is thus given by a family of module morphisms (of degree zero), indexed n times over the respective object type as described in [Rem. 5.6](#);
- the relation on the initial object is not defined via the formula of [Disp. \(5.4.1\)](#), but directly through an inductive type, cf. [Code 9.9](#), and various closures, cf. [Code 9.10](#).

9.1. Representations of PCF

In this section we explain the formalization of representations of semantic PCF. According to [Def. 5.10](#) and [Def. 5.20](#), such a representation consists of

1. a representation of the types of PCF (in a Coq type U), cf. [Ex. 3.4](#),
2. a relative monad P over the functor Δ^U (in the formalization: $\text{IDelta } U$) and
3. representations of the arities of PCF (cf. [Ex. 3.48](#)), i.e. morphisms of P –modules with suitable source and target modules such that
4. the inequations defining the reduction rules of PCF are satisfied.

A representation of PCF should be a “bundle”, i.e. a record type, whose components — or “fields” — are these 4 items. In order to ease the definitions, we first define what a representation of the term signature of PCF in a monad P is, in the presence of an S_{PCF} –monad (cf. [Def. 5.1](#)). Unfolding the definitions, we suppose given a type Sorts , a relative monad P over $\text{IDelta } \text{Sorts}$ and three operations on Sorts : a binary function Arrow — denoted by an infix “ $\sim\sim>$ ” — and two constants Bool and Nat .

Variable $\text{Sorts} : \text{Type}$.

Variable $P : \text{RMond} (\text{IDelta Sorts})$.

Variable $\text{Arrow} : \text{Sorts} \rightarrow \text{Sorts} \rightarrow \text{Sorts}$.

Variable $\text{Bool} : \text{Sorts}$.

Variable $\text{Nat} : \text{Sorts}$.

Notation $"a \sim\sim> b" := (\text{Arrow } a \ b)$ (at level 60, **right** associativity).

In this context, a representation of PCF is given by a bunch of module morphisms satisfying some conditions. We split the definition into smaller pieces. Note that $M[t]$ denotes the fibre module of module M with respect to t , and $d \ M \ // \ u$ denotes derivation of module M with respect to u . The module denoted by a star $*$ is the terminal module, which is the constant singleton module.

9.1 Code (1–Signature of PCF):

```
Class PCFPO_rep_struct := {
  app : forall u v, (P[u ~~~> v]) x (P[u]) ----> P[v];
  abs : forall u v, (d P // u)[v] ----> P[u ~~~> v];
  rec : forall t, P[t ~~~> t] ----> P[t];
  tttt : * ----> P[Bool];
  ffff : * ----> P[Bool];
  nats : forall m:nat, * ----> P[Nat];
  Succ : * ----> P[Nat ~~~> Nat];
  Pred : * ----> P[Nat ~~~> Nat];
  Zero : * ----> P[Nat ~~~> Bool];
  CondN : * ----> P[Bool ~~~> Nat ~~~> Nat ~~~> Nat];
  CondB : * ----> P[Bool ~~~> Bool ~~~> Bool ~~~> Bool];
  bottom : forall t, * ----> P[t];
  ...
}
```

These module morphisms are subject to some inequations specifying the reduction rules of [Sect. A.2](#), or, equivalently, [Ex. 5.19](#). The beta rule reads as

9.2 Code (Beta Rule for Representations of PCF):

```
beta_red : forall r s V y z, app r s V (abs r s V y, z) << y[* := z] ;
...
```

where $y[* := z]$ is the substitution of the freshest variable (cf. [Def. 2.111](#)) as a special case of simultaneous monadic substitution. The rule for the fixed point operator says that $Y(f) \rightsquigarrow f(Y(f))$:

9.3 Code (Inequation for Fixedpoint Operator):

```
Rec_A : forall V t g, rec _ _ g << app t t V (g, rec _ _ g) ;
...
```


The other inequations concern the arithmetic and logical constants of PCF. Firstly, we have that the conditionals reduce according to the truth value they are applied to:

9.4 Code (Logic Inequations of PCF Representations):

```

CondN_t: forall V n m,
  app _ _ _ (app _ _ _ (app _ _ _ (CondN V tt, tttt _ tt), n), m) << n ;
CondN_f: forall V n m,
  app _ _ _ (app _ _ _ (app _ _ _ (CondN V tt, ffff _ tt), n), m) << m ;
CondB_t: forall V u v,
  app _ _ _ (app _ _ _ (app _ _ _ (CondB V tt, tttt _ tt), u), v) << u ;
CondB_f: forall V u v,
  app _ _ _ (app _ _ _ (app _ _ _ (CondB V tt, ffff _ tt), u), v) << v ;
...

```

Furthermore, we have that $\text{succ}(n)$ reduces to $n + 1$ (which in Coq is written $S\ n$), reduction of the zero? predicate according to whether its argument is zero or not, and that the predecessor is post-inverse to the successor function:

9.5 Code (Arithmetic Inequations of PCF Representations):

```

Succ_red: forall V n,
  app _ _ _ (Succ V tt, nats n _ tt) << nats (S n) _ tt ;
Zero_t: forall V,
  app _ _ _ (Zero V tt, nats 0 _ tt) << tttt _ tt ;
Zero_f: forall V n,
  app _ _ _ (Zero V tt, nats (S n) _ tt) << ffff _ tt ;
Pred_Succ: forall V n,
  app _ _ _ (Pred V tt, app _ _ _ (Succ V tt, nats n _ tt)) << nats n _ tt ;
Pred_Z: forall V,
  app _ _ _ (Pred V tt, nats 0 _ tt) << nats 0 _ tt }.

```

Unfortunately, at this stage of the definition, we were not able to introduce a more convenient notation for application, neither to omit the arguments denoted by an underscore as instances of implicit arguments. After abstracting over the section variables we package all of this into a record type:

```

Record PCFPO_rep := {
  Sorts : Type;
  Arrow : Sorts -> Sorts -> Sorts;
  Bool : Sorts ;

```

```
Nat : Sorts ;
pcf_rep_monad :> RMonad (IDelta Sorts);
pcf_rep_struct :> PCFPO_rep_struct pcf_rep_monad Arrow Bool Nat }.
```

Notation "a $\sim\sim>$ b" := (Arrow a b) (at level 60, [right](#) associativity).

The type PCFPO_rep later constitutes the type of objects of the category of representations of semantic PCF.

9.2. Morphisms of Representations

A morphism of representations (cf. [Def. 5.11](#)) is built from a morphism g of *type* representations and a colax monad morphism over the retyping functor associated to the map g . The implementation of retyping is explained in [Code 6.6](#). In the particular case of PCF, a morphism of representations from P to R consists of a morphism of representations of the types of PCF — with underlying map `Sorts_map` — and a colax morphism of relative monads which makes commute the diagrams of the form given in [Def. 5.11](#). We first define the diagrams we expect to commute, before packaging everything into a record type of morphisms. The context is given by the following declarations:

```
Variables P R : PCFPO_rep.
Variable Sorts_map : Sorts P -> Sorts R.
Hypothesis HArrow : forall u v, Sorts_map (u  $\sim\sim>$  v) = Sorts_map u  $\sim\sim>$ 
Sorts_map v.
Hypothesis HBool : Sorts_map (Bool _) = Bool _ .
Hypothesis HNat : Sorts_map (Nat _) = Nat _ .

Variable f : colax_RMonad_Hom P R
  (G1:=RETYPE (fun t => Sorts_map t))
  (G2:=RETYPE_PO (fun t => Sorts_map t))
  (RT_NT (fun t => Sorts_map t)).
```

We explain the commutative diagrams of [Def. 5.11](#) for some of the arities. For the successor arity we ask the following diagram to commute:

9.6 Code (Commutative Diagram for Successor Arity):

```
Program Definition Succ_hom' :=
  Succ ;; f [(Nat  $\sim\sim>$  Nat)] ;; Fib_eq_RMod _ _ ;; IsoPF
  ==
  * ---->* ;; f * * Succ.
```

Here the morphism `Succ` refers to the representation of the successor arity either of P (the first appearance) or R (the second appearance) — Coq is able to figure this out itself.

The domain of the successor is given by the terminal module $*$. Accordingly, we have that $\text{dom}(\text{Succ}, f)$ is the trivial module morphism with domain and codomain given by the terminal module. We denote this module morphism by $* \dashrightarrow *$. The codomain is given as the fibre of f of type $\iota \Rightarrow \iota$. The two remaining module morphisms are isomorphisms which do not appear in the informal description. The isomorphism IsoPF is needed to permute fibre with pullback (cf. [Lem. 2.108](#)). The morphism $\text{Fib_eq_RMod } M \ H$ takes a module M and a proof H of equality of two object types as arguments, say, $H : u = v$. Its output is an isomorphism $M[u] \dashrightarrow M[v]$. Here the proof is of type

$$\text{Sorts_map } (\text{Nat} \rightsquigarrow \text{Nat}) = \text{Sorts_map } \text{Nat} \rightsquigarrow \text{Sorts_map } \text{Nat}$$

and Coq is able to figure out the proof itself. We expand on this kind of modules in [Sect. 9.3](#) The diagram for application uses the product of module morphisms, denoted by an infix X :

9.7 Code (Commutative Diagram for Application Arity):

```
Program Definition app_hom' := forall u v,
  app u v ;; f [( _ )] ;; IsoPF
  ==
  (f [(u ~> v)] ;; Fib_eq_RMod _ (HArrow _ _)) ;; IsoPF )
  X
  (f [(u)] ;; IsoPF ) ;;
  IsoXP ;; f * * (app _ _ ).
```

In addition to the already encountered isomorphism IsoPF we have to insert an isomorphism IsoXP which permutes pullback and product (cf. [Lem. 2.106](#)). As a last example, we present the property for the abstraction:

9.8 Code (Commutative Diagram for Abstraction Arity):

```
Program Definition abs_hom' := forall u v,
  abs u v ;; f [( _ )]
  ==
  DerFib_RMod_Hom _ _ _ ;; IsoPF ;;
  f * * (abs ( _ u) ( _ v)) ;; IsoFP ;;
  Fib_eq_RMod _ (eq_sym (HArrow _ _)) .
```

Here the module morphism $\text{DerFib_RMod_Hom } f \ u \ v$ corresponds to the morphism $\text{dom}(\text{Abs}(u, v), f) = [f^u]_v$, and IsoFP permutes fibre with pullback, just like its sibling IsoPF , but the other way round.

We bundle all those properties into a type class:

```
Class PCFPO_rep_Hom_struct := {
  CondB_hom : CondB_hom' ;
  CondN_hom : CondN_hom' ;
```

```

Pred_hom : Pred_hom' ;
Zero_hom : Zero_hom' ;
Succ_hom : Succ_hom' ;
fff_hom : fff_hom' ;
ttt_hom : ttt_hom' ;
bottom_hom : bottom_hom' ;
nats_hom : nats_hom' ;
app_hom : app_hom' ;
rec_hom : rec_hom' ;
abs_hom : abs_hom' }.

```

Similarly to what we did for representations, we abstract over the section variables and define a record type of morphisms of representations from P to R :

```

Record PCFPO_rep_Hom := {
  Sorts_map : Sorts P -> Sorts R ;
  HArrow : forall u v, Sorts_map (u ~> v) = Sorts_map u ~> Sorts_map v;
  HNat : Sorts_map (Nat _) = Nat R ;
  HBool : Sorts_map (Bool _) = Bool R ;
  rep_hom_monad :> colax_RMonad_Hom P R (RT_NT Sorts_map);
  rep_colax_Hom_monad_struct :> PCFPO_rep_Hom_struct
    HArrow HBool HNat rep_hom_monad }.

```

9.3. Digression on Equal Fibre Modules in Coq

Suppose Q is a relative monad on some functor $F : \mathcal{C} \rightarrow \mathcal{D}$ and M is a Q -module with codomain Pre^T . Let $u, t \in T$ and suppose given a proof H of the proposition $u = t$. We can now prove $[M]_u = [M]_t$, but unfortunately this is not sufficient for composing a morphism with codomain $[M]_u$ with one whose domain is $[M]_t$ in Coq (cf. [Sect. 6.3.1](#)). Indeed, the problem we encounter here is even worse than that of permutation of pullback with fibre, derivation and products (see e.g. [Sect. 7.1.2](#)), since not even the carriers of $[M]_u$ and $[M]_t$ are convertible. This means that the isomorphism we have to insert does not even allow for an underlying family of *identity* maps as carriers, but instead is a transport of the form eq_rect .

In more detail, the carrier of M is a map from the objects of \mathcal{C} to Pre^T , that is, for each $c \in \mathcal{C}$, its image $Mc \in \text{Pre}^T$ is basically a dependent type (with some structure). The fibre is then simply computed by application. The carrier of a module morphism $\rho : [M]_u \rightarrow [M]_t$ thus consists of a family of maps of sets indexed by objects $c \in \mathcal{C}$,

$$\rho_c : M(c)(u) \rightarrow M(c)(t) .$$

In intensional type theory, we have an *explicit* cast operator `eq_rect` which allows the definition of precisely such a map:

Check `eq_rect`.
`eq_rect`
`: forall (A : Type) (x : A) (P : A -> Type),`
`P x -> forall y : A, x = y -> P y`

Note that this operator is equivalent to the operator `J` in Hofmann’s PhD thesis [Hof95], whose typing rule is called ID-ELIM-J.

Here we instantiate `A` by set of object types `T` and the dependent type `P` by `M(c)`, allowing us to define a map transport from `M(c)(u)` to `M(c)(t)`:

Variable `T : Type`.
 Variables `u t : T`.
 Variable `M : RMOD Q (IPO T)`.
 Hypothesis `H : u = t`.
 Definition `transport (c : C) : M c u -> M c t :=`
`fun (s : M c u) => eq_rect u (fun t : T => (M c) t) s t H.`

Fortunately it is possible to get rid of the transport via a computation rule equivalent to a rule named ID-COMP in Hofmann’s thesis. In Coq this rule says that the term

`eq_rect u P a a eq_refl`

reduces to `a` — and thus in particular is provably equal to `a` — the term `a` itself. Thus a considerable part of proof code in the following is about elimination of explicit casts. Indeed, the scheme is as follows: we start with a goal

----- (1/1)
 G

such that `G` contains a subterm `eq_rect u P a b H`, i.e. with `H : a = b`. We then generalize `H`, yielding the goal

----- (1/1)
 forall H : a = b, G

Now rewriting with a proof of `a = b` (using a copy of `H`) turns the goal into

----- (1/1)
 forall H : a = a, G

After introducing `H`, we can rewrite `H` in the goal into `eq_refl` using the axiom `UIP_refl` which says that any proof of `a = a` is equal to `eq_refl`. Thus the goal `G` contains the subterm `eq_rect u P a a eq_refl`, which simplifies to `a` — the transport has disappeared. Note that for the rewrite of `forall H : a = b` into `forall H : a = a` in the goal, many other terms from the context have to be generalized, as well as structures broken into their

constituent pieces, in order to obtain sufficient flexibility in the goal for the rewrite to result in a well-typed term.

9.4. Equality of Morphisms, Category of Representations

We have already seen how some definitions that are trivial in informal mathematics, turn into something awful in intensional type theory. Equality of morphisms of representations is another such definition. Informally, two such morphisms $a, c : P \rightarrow R$ of representations are equal if

1. their map of object types f_a and f_c (Sorts_map) are equal and
2. their underlying colax morphism of monads — also called a and c — are equal.

In our formalization, the second condition is not even directly expressable, since these monad morphisms do not have the same type: we have, for a context $V \in \text{Set}^P$,

$$a_V : \vec{f}_a(PV) \rightarrow R(\vec{f}_a V)$$

and

$$c_V : \vec{f}_c(PV) \rightarrow R(\vec{f}_c V) .$$

where Set^P is a notation for contexts typed over the set of object types the representation P comes with, formally the type $\text{Sorts } P$. We can only compare a_V to c_V by composing each of them with a suitable transport transp again, yielding morphisms

$$R(\text{transp}) \circ a_V : \vec{f}_a(PV) \rightarrow R(\vec{f}_a V) \rightarrow R(\vec{f}_c V)$$

and

$$c_V \circ \text{transp}' : \vec{f}_c(PV) \rightarrow \vec{f}_c(PV) \rightarrow R(\vec{f}_c V) .$$

As before, for equal fibres $[M]_u$ and $[M]_t$ with $u = t$, the carriers of those transports transp and transp' are terms of the form $\text{eq_rect } _ _ _ H$, where H is a proof term which depends on the proof of

`forall x : Sorts P, Sorts_map c x = Sorts_map a x`

of the first condition. Altogether, the definition of equality of morphisms of representations is given by the following inductive proposition:

Inductive `eq_Rep (P R : PCFPO_rep) : relation (PCFPO_rep_Hom P R) :=`
`| eq_rep : forall (a c : PCFPO_rep_Hom P R),`
`forall H : (forall t, Sorts_map c t = Sorts_map a t),`

$$\begin{aligned}
 & (\text{forall } V, a \ V \ ; \ \text{rlift } R \ (\text{Transp } H \ V) \\
 & \quad == \\
 & \quad \text{Transp_ord } H \ (P \ V) \ ; \ c \ V \) \ \rightarrow \text{eq_Rep } a \ c.
 \end{aligned}$$

The formal proof that the relation thus defined is an equivalence is inadequately long when compared to its mathematical complexity, due to the transport elimination.

Composition of representations is done by composing the underlying maps of sorts, as well as composing the underlying monad morphisms pointwise. Again, this operation, which is trivial from a mathematical point of view, yields a difficulty in the formalization, due to the fact that in the formalization

$$\vec{g}(\vec{f}V) \neq (g \circ f)V.$$

More precisely, suppose given two morphisms of representations $a : P \rightarrow Q$ and $b : Q \rightarrow R$, given by families of morphisms indexed by V resp. W ,

$$\begin{aligned}
 a_V & : \widetilde{P}V^a \rightarrow Q(\widetilde{V}^a) \quad \text{and} \\
 b_W & : \widetilde{Q}W^b \rightarrow R(\widetilde{W}^b),
 \end{aligned}$$

where we write \widetilde{V}^a for $\vec{f}_a V$. The monad morphism underlying the composite morphism of representations is given by the following definition:

$$\begin{array}{ccc}
 \widetilde{P}V^{b \circ a} & \xrightarrow{b \circ a_V} & R(\widetilde{V}^{b \circ a}) \\
 \text{match} \downarrow & & \uparrow R(\cong) \\
 P V & & R(\widetilde{V}^a{}^b) \\
 \text{ctype} \downarrow & & \uparrow b_{\widetilde{V}^a} \\
 \widetilde{P}V^a & \xrightarrow{a_V} Q(\widetilde{V}^a) \xrightarrow{\text{ctype}} \widetilde{Q}(\widetilde{V}^a)^b
 \end{array}$$

or, in Coq code,

```

Definition comp_rep_car : (forall c : ITYPE U,
  RETYPE (fun t => f' (f t)) (P c) ---->
  R ((RETYPE (fun t => f' (f t))) c)) :=
fun (V : ITYPE U) t (y : retype (fun t => f' (f t)) (P V) t) =>
match y with ctype _ z =>
  lift (M:=R) (double_retype_1 (f:=f) (f':=f') (V:=V)) _
    (b _ _ (ctype (fun t => f' t)
      (a _ _ (ctype (fun t => f t) z )))))
end.
    
```

where `double_retype_1` denotes the isomorphism in the upper right corner. The proof of the commutative diagrams for the composite monad morphism is lengthy due to the number of arities of the signature of PCF. Definition of the identity morphisms is routine, and in the end we define the category of representations of semantic PCF:

```
Program Instance REP_s :
  Cat_struct (obj := PCFPO_rep) (PCFPO_rep_Hom) := {
    mor_oid P R := eq_Rep_oid P R ;
    id R := Rep_id R ;
    comp a b c f g := Rep_comp f g }.
```

9.5. One Particular Representation

We define a particular representation, which we later prove to be initial. First of all, the set of object types of PCF is given as follows:

```
Module PCF.
Inductive Sorts :=
  | Nat : Sorts
  | Bool : Sorts
  | Arrow : Sorts -> Sorts -> Sorts.
End PCF.
```

For this section we introduce some notations:

```
Notation "'TY'" := PCF.Sort.
Notation "'Bool'" := PCF.Bool.
Notation "'Nat'" := PCF.Nat.
Notation "'IT'" := (ITYPE TY).
Notation "a '~>' b" := (PCF.Arrow a b) (at level 69, right associativity).
```

We specify the set of PCF constants through the following inductive type, indexed by the sorts of PCF:

```
Inductive Consts : TY -> Type :=
  | Nats : nat -> Consts Nat
  | ttt : Consts Bool
  | fff : Consts Bool
  | succ : Consts (Nat ~> Nat)
  | preds : Consts (Nat ~> Nat)
  | zero : Consts (Nat ~> Bool)
  | condN : Consts (Bool ~> Nat ~> Nat ~> Nat)
  | condB : Consts (Bool ~> Bool ~> Bool ~> Bool).
```


The set family of terms of PCF is given by an inductive family, parametrized by a context V and indexed by object types:

Inductive PCF ($V: TY \rightarrow \text{Type}$) : $TY \rightarrow \text{Type} :=$
 | Bottom: forall t, PCF V t
 | Const : forall t, Consts t \rightarrow PCF V t
 | Var : forall t, V t \rightarrow PCF V t
 | App : forall t s, PCF V (s \sim t) \rightarrow PCF V s \rightarrow PCF V t
 | Lam : forall t s, PCF (opt t V) s \rightarrow PCF V (t \sim s)
 | Rec : forall t, PCF V (t \sim t) \rightarrow PCF V t.
Notation "a @ b" := (App a b)(at level 43, left associativity).
Notation "M '" := (Const _ M) (at level 15).

Monadic substitution is defined recursively on terms:

Fixpoint subst ($V W: TY \rightarrow \text{Type}$)(f: forall t, V t \rightarrow PCF W t)
 (t : TY)(v : PCF V t) : PCF W t :=
 match v with
 | Bottom t => Bottom W t
 | c ' => c '
 | Var t v => f t v
 | u @ v => u >>= f @ v >>= f
 | Lam t s u => Lam (u >>= shift f)
 | Rec t u => Rec (u >>= f)
 end

where "y >>= f" := (@subst _ _ f _ y).

Here shift f is the substitution map f extended to account for an extended context under the binder Lam. It is equal to the shifted map of Def. 2.102.

Finally, we define a relation on the terms of type PCF via the inductive definition

9.9 Code (Reduction Rules for PCF):

Inductive eval ($V : IT$): forall t, relation (PCF V t) :=
 | app_abs : forall (s t:TY) (M: PCF (opt s V) t) N,
 eval (Lam M @ N) (M [* := N])
 | condN_t: forall n m, eval (condN ' @ ttt ' @ n @ m) n
 | condN_f: forall n m, eval (condN ' @ fff ' @ n @ m) m
 | condB_t: forall u v, eval (condB ' @ ttt ' @ u @ v) u
 | condB_f: forall u v, eval (condB ' @ fff ' @ u @ v) v
 | succ_red: forall n, eval (succ ' @ Nats n ') (Nats (S n) ')
 | zero_t: eval (zero ' @ Nats 0 ') (ttt ')
 | zero_f: forall n, eval (zero ' @ Nats (S n)') (fff ')
 | pred_Succ: forall n, eval (preds ' @ (succ ' @ Nats n ')) (Nats n ')

9. A Faithful Translation of PCF to ULC

```
| pred_z: eval (preds ' @ Nats 0 ') (Nats 0 ')
| rec_a : forall t g, eval (Rec g) (g @ (Rec (t:=t) g)).
```

which we then propagate into subterms (cf. [Code 9.10](#)) and close with respect to transitivity and reflexivity:

9.10 Code (Propagation of Reductions into Subterms):

Reserved Notation "x :> y" (at level 70).

Variable rel : forall (V:IT) t, relation (PCF V t).

Inductive propag (V: IT) : forall t, relation (PCF V t) :=

```
| relorig : forall t (v v' : PCF V t), rel v v' -> v :> v'
| relApp1: forall s t (M M' : PCF V (s ~> t)) N, M :> M' -> M @ N :> M' @ N
| relApp2: forall s t (M : PCF V (s ~> t)) N N', N :> N' -> M @ N :> M @ N'
| relLam: forall s t (M M':PCF (opt s V) t), M :> M' -> Lam M :> Lam M'
| relRec: forall t (M M' : PCF V (t ~> t)), M :> M' -> Rec M :> Rec M'
```

where "x :> y" := (@propag _ _ x y).

The data thus defined constitutes a relative monad PCFEM on the functor $\Delta^{T_{PCF}}$ (IDelta TY). We omit the details.

Now we need to define a suitable morphism (resp. family of morphisms) of PCFEM-modules for any arity (of higher degree). Let α be any such arity, for instance the arity App. We need to verify two things:

1. we show that the constructor of PCF which corresponds to α is monotone with respect to the order on PCFEM. For instance, we show that for any two terms $r s:TY$ and any $V : IDelta TY$, the function

```
fun y => App (fst y) (snd y): PCFEM V (r~>s) x PCFEM V r -> PCFEM
  V s
```

is monotone.

2. We show that the monadic substitution defined above distributes over the constructor in the sense of [Ex. 2.74](#), i.e. we prove that the constructor is the carrier of a *module* morphism.

All of these are very straightforward proofs, resulting in a representation PCFE_rep of semantic PCF:

Program Instance PCFE_rep_struct :

```
PCFPO_rep_struct PCFEM PCF.arrow PCF.Bool PCF.Nat := {
  app r s := PCFApp r s;
  abs r s := PCFAbs r s;
  rec t := PCFRec t ;
  tttt := PCFconsts ttt ;
```

```

ffff := PCFconsts fff;
Succ := PCFconsts succ;
Pred := PCFconsts preds;
CondN := PCFconsts condN;
CondB := PCFconsts condB;
Zero := PCFconsts zero ;
nats m := PCFconsts (Nats m);
bottom t := PCFbottom t }.

```

Definition `PCFE_rep : PCFPO_rep := Build_PCFPO_rep PCFE_rep_struct.`

Note that in the instance declaration `PCFE_rep_struct`, the **Program** framework proves automatically the properties of [Code 9.2](#), [9.3](#), [9.4](#) and [9.5](#).

9.6. Initiality

In this section we define a morphism of representations from `PCFE_rep` to any representation `R : PCFPO_rep`. At first we need to define a map between the underlying sorts, that is, a map `Sorts PCFE_rep -> Sorts R`. In short, each PCF type goes to its representation in `R`:

```

Fixpoint Init_Sorts_map (t : Sorts PCFE_rep) : Sorts R :=
  match t with
  | PCF.Nat => Nat R
  | PCF.Bool => Bool R
  | u ~> v => (Init_Sorts_map u) ~> (Init_Sorts_map v)
  end.

```

The function `init` is the carrier of what will later be proved to be the initial morphism to the representation `R`. It maps each constructor of PCF recursively to its counterpart in the representation `R`:

```

Fixpoint init V t (v : PCF V t) :
  R (retype (fun t0 => Init_Sorts_map t0) V) (Init_Sorts_map t) :=
  match v with
  | Var t v => rweta R _ _ (ctype _ v)
  | u @ v => app _ _ (init u, init v)
  | Lam _ _ v => abs _ _ (rlift R
    (@der_comm TY (Sorts R) (fun t => Init_Sorts_map t) _ V) _ (init v
    ))
  | Rec _ v => rec _ _ (init v)
  | Bottom _ => bottom _ _ tt
  | y ' => match y in Consts t1 return

```

```

R (retype (fun t2 => Init_Sorts_map t2) V) (Init_Sorts_map t1)
  with
  | Nats m => nats m _ tt
  | succ => Succ _ tt
  | condN => CondN _ tt
  | condB => CondB _ tt
  | zero => Zero _ tt
  | ttt => tttt _ tt
  | fff => ffff _ tt
  | preds => Pred _ tt
  end.
end.

```

We write i_V for $\text{init } V$ and g for Init_Sorts_map . Note that $i_V : \text{PCF}(V) \rightarrow g^*(R(\vec{g}V))$ really is the image of the initial morphism under the adjunction φ of Def. 2.22. Intuitively, passing from $\text{init } V = i_V$ to its adjunct $\varphi^{-1}(i_V)$ is done by precomposing with pattern matching on the constructor ctype (cf. Rem. 2.25). We informally denote $\varphi^{-1}(i_V)$ by $\text{init } V \circ \text{match}$.

The map init is compatible with renaming and substitution in PCF and R, respectively, in a sense made precise by the following two lemmas. The first lemma states that, for any morphism $f : V \rightarrow W$ in $\text{Set}^{T_{\text{PCF}}}$, the following diagram commutes:

$$\begin{array}{ccc}
 \text{PCF}(V) & \xrightarrow{\text{PCF}(f)} & \text{PCF}(W) \\
 \text{init } V \downarrow & & \downarrow \text{init } W \\
 g^*R(\vec{g}V) & \xrightarrow{g^*R(g^*f)} & g^*R(\vec{g}W).
 \end{array}$$

Lemma $\text{init_lift } (V : \text{IT}) \text{ } t \text{ } (y : \text{PCF } V \text{ } t) \text{ } W \text{ } (f : V \dashrightarrow W) :$
 $\text{init } (y \text{ } // \text{ } - \text{ } f) = \text{rlift } R \text{ } (\text{retype_map } f) \text{ } _ \text{ } (\text{init } y).$

The next commutative diagram concerns substitution; for any $f : V \rightarrow \text{PCF}(W)$, the diagram obtained by applying φ to the diagram given in Disp. (5.4.4) — i.e. the diagram corresponding to Disp. (5.4.5) —, commutes:

$$\begin{array}{ccc}
 \text{PCF}(V) & \xrightarrow{\sigma^{\text{PCF}(f)}} & \text{PCF}(W) \\
 \text{init } V \downarrow & & \downarrow \text{init } W \\
 g^*R(\vec{V}) & \xrightarrow{g^*\sigma^R(\varphi^{-1}(\text{init } W) \circ (g^*f))} & g^*R(\vec{W}).
 \end{array}$$

In Coq the lemma init_subst proves commutativity of this latter diagram:

Lemma `init_subst V t (y : PCF V t) W (f : IDelta _ V ----> PCFE W):`
`init (y >>= f) =`
`rkleisli (RMonad_struct := R)`
`(SM_ind (V:= retype (fun t => _ t) V)`
`(W:= R (retype (fun t => _ t) W))`
`(fun t v => match v with ctype t p => init (f t p) end))`
`_ (init y).`

This latter lemma establishes almost the commutative diagram for the family $\varphi^{-1}(i_V)$ to constitute a (colax) *monad* morphism, which reads as follows:

$$\begin{array}{ccc}
 \vec{g}(\text{PCF}(V)) & \xrightarrow{\vec{g}(\sigma^{\text{PCF}}(f))} & \vec{g}(\text{PCF}(W)) \\
 \text{init } V \circ \text{match} \downarrow & & \downarrow \text{init } W \circ \text{match} \\
 R(\vec{g}V) & \xrightarrow{\sigma^R(\text{init} \circ \text{match} \circ (\vec{g}f))} & R(\vec{g}W).
 \end{array} \tag{9.6.1}$$

Before we can actually build a monad morphism with carrier map `init V ◦ match`, we need to verify that `init` — and thus its adjunct — is monotone. We do this in 3 steps, corresponding to the 3 steps in which we built up the preorder on the terms of PCF:

1. `init` monotone with respect to the relation `eval` (cf. [Code 9.9](#)):

Lemma `init_eval V t (v v' : PCF V t) : eval v v' -> init v <<< init v'.`

2. `init` monotone with respect to the propagation into subterms of `eval`;

Lemma `init_eval_star V t (y z : PCF V t) : eval_star y z -> init y <<< init z.`

3. `init` monotone with respect to reflexive and transitive closure of above relation.

Lemma `init_mono c t (y z : PCFE c t) : y <<< z -> init y <<< init z.`

We now have all the ingredients to define the initial morphism from PCF to R. As already indicated by the diagram [Disp. \(9.6.1\)](#), its carrier is not given by just the map `init`, since this map does not have the right type: its domain is given, for any context $V \in \text{Set}^{T_{\text{PCF}}}$, by `PCF(V)` and not, as needed, by $\vec{g}(\text{PCF}(V))$. We thus precompose with pattern matching in order to pass to its adjunct: for any context V , the carrier of the initial morphism is given by

```

fun t y => match y with
| ctype _ p => init p
end
: retype _ (PCF V) ----> R (retype _ W)

```

We recall that the constructor `ctype` is the carrier of the natural transformation of the same name of [Rem. 2.23](#), and that precomposing with pattern matching corresponds to specifying maps on a coproduct via its universal property.

Putting the pieces together, we obtain a morphism of representations of semantic PCF:

Definition `initR : PCFPO_rep_Hom PCFE_rep R :=
Build_PCFPO_rep_Hom initR_s.`

Uniqueness is proved in the following lemma:

Lemma `initR_unique : forall g : PCFE_rep ----> R, g == initR.`

The proof consists of two steps: first, one has to show that the translation of *sorts* coincide. Since the source of this translation is an inductive type — the initial representation of the signature of [Ex. 3.4](#) — this proof is done by induction. Afterwards the translations of terms are proved to be equal. The proof is done by induction on terms of PCF. It makes essentially use of the commutative diagrams (cf. [Def. 5.11](#)) which we exemplarily presented for the arities of successor ([Code 9.6](#)), application ([Code 9.7](#)) and abstraction ([Code 9.8](#)). Finally we can declare an instance of `Initial` for the category `REP` of representations:

Instance `PCF_initial : Initial REP := {
Init := PCFE_rep ;
InitMor R := initR R ;
InitMorUnique R := @initR_unique R }.`

Checking the axioms used for the proof of initiality (and its dependencies) yields the use of non-dependent functional extensionality (applied to the translations of sorts) and uniqueness of identity proofs, which in the Coq standard library is implemented as a consequence of another — logically equivalent — axiom `eq_rect_eq`:

Print Assumptions `PCF_initial`.

Axioms:

```
CatSem.AXIOMS.functional_extensionality.functional_extensionality :  
  forall (A B : Type) (f g : A -> B),  
    (forall x : A, f x = g x) -> f = g  
Eq_rect_eq.eq_rect_eq : forall (U : Type) (p : U) (Q : U -> Type)  
  (x : Q p) (h : p = p), x = eq_rect p Q x p h
```

9.7. A Representation of PCF in the Untyped Lambda Calculus

We use the iteration principle explained in [Rem. 5.23](#) in order to specify a translation from PCF to the untyped lambda calculus which is compatible with reduction in the

source and target. According to the principle, it is sufficient to define a representation of PCF in the relative monad of the lambda calculus (cf. Exs. 1.2 and 2.85) and to verify that this representation satisfies the inequations of Fig. A.4, formalized in the Coq code snippets 9.2, 9.3, 9.4 and 9.5. The first task, specifying a representation of the types of PCF, in the singleton set of types of ULC, is trivial. We furthermore specify representations of the term arities of PCF, presented in Code 9.1, by giving an instance of the corresponding type class.

```

Program Instance PCF_ULC_rep_s :
  PCFPO_rep_struct (Sorts:=unit) ULCBETAM (fun _ _ => tt) tt tt := {
    app r s := ulc_app r s;
    abs r s := ulc_abs r s;
    rec t := ulc_rec t ;
    tttt := ulc_ttt ;
    ffff := ulc_fff ;
    nats m := ulc_N m ;
    Succ := ulc_succ ;
    CondB := ulc_condb ;
    CondN := ulc_condn ;
    bottom t := ulc_bottom t ;
    Zero := ulc_zero ;
    Pred := ulc_pred }.

```

Before taking a closer look at the module morphisms we specify in order to represent the arities of PCF, we note that in the above instance declaration, we have not given the proofs corresponding to code snippets 9.2 to 9.5. In the terms of Rem. 5.23, we have not completed the third task, the verification that the given representation satisfies the inequations. The **Program** feature we use during the above instance declaration is able to detect that the fields called `beta_red`, `rec_A`, etc., are missing, and enters into interactive proof mode to allow us to fill in each of the missing fields.

We now take a look at some of the lambda terms representing arities of PCF. The carrier of the representations `ulc_app` is the application of lambda calculus, of course, and similar for `ulc_abs`. Here the parameters `r` and `s` vary over terms of type `unit`, the type of sorts underlying this representation. We use an infix application and a de Bruijn notation instead of the more abstract notation of nested data types:

```

Notation "a @ b" := (App a b) (at level 42, left associativity).
Notation "'1'" := (Var None) (at level 33).
Notation "'2'" := (Var (Some None)) (at level 24).

```

The truth values **T** and **F** are represented by

```

Eval compute in ULC_True.
  = Abs (Abs 2)

```

9. A Faithful Translation of PCF to ULC

Eval compute in ULC_False.
 $= \text{Abs } (\text{Abs } 1)$

Natural numbers are given in Church style, the successor function is given by the term $\lambda n f x. f(n f x)$. The predecessor is represented by the constant

$$\lambda n f x. n (\lambda g h. h(g f)) (\lambda u. x) (\lambda u. u),$$

and the test for zero is represented by $\lambda n. n(\lambda x. F)T$, where F and T are the lambda terms representing **F** and **T**, respectively.

Eval compute in ULC_Nat 0.
 $= \text{Abs } (\text{Abs } 1)$

Eval compute in ULC_Nat 2.
 $= \text{Abs } (\text{Abs } (2 @ (\text{Abs } (\text{Abs } (2 @ (\text{Abs } (\text{Abs } 1) @ 2 @ 1))) @ 2 @ 1)))$

Eval compute in ULCsucc.
 $= \text{Abs } (\text{Abs } (\text{Abs } (2 @ (3 @ 2 @ 1))))$

Eval compute in ULC_pred.
 $= \text{Abs } (\text{Abs } (\text{Abs } (3 @ \text{Abs } (\text{Abs } (1 @ (2 @ 4))) @ \text{Abs } 2 @ \text{Abs } 1)))$

Eval compute in ULC_zero.
 $= \text{Abs } (1 @ \text{Abs } (\text{Abs } (\text{Abs } 1)) @ \text{Abs } (\text{Abs } 2))$

The conditional is represented by the lambda term $\lambda p a b. p a b$:

Eval compute in ULC_cond.
 $= \text{Abs } (\text{Abs } (\text{Abs } (3 @ 2 @ 1)))$

The constant arity \perp_A is represented by Ω :

Eval compute in ULC_omega.
 $= \text{Abs } (1 @ 1) @ \text{Abs } (1 @ 1)$

The fixed point operator **Fix** (rec) is represented by the *Turing* fixed-point combinator, that is, the lambda term

Eval compute in ULC_theta.
 $= \text{Abs } (\text{Abs } (1 @ (2 @ 2 @ 1))) @ \text{Abs } (\text{Abs } (1 @ (2 @ 2 @ 1)))$

The reason why we use the Turing operator instead of, say, the combinator **Y**,

Eval compute in ULC_Y.
 $= \text{Abs } (\text{Abs } (2 @ (1 @ 1)) @ \text{Abs } (2 @ (1 @ 1)))$

is that the latter does not have a property that is crucial for us: It is

$$\Theta(f) \rightsquigarrow^* f (\Theta(f))$$

but only

$$Y(f) \stackrel{*}{\leftrightarrow} f (Y(f))$$

via a common reduct. Thus if we would attempt to represent the arity `rec` by the fixed-point combinator `Y`, we would not be able to prove the condition expressed in [Code 9.3](#). A way to allow for the use of `Y` as representation of `rec` would be to consider *symmetric* relations on terms, e.g., relative monads into a category of setoids.

As a final remark, we emphasize that while reduction is given as a relation in our formalization, and as such is not computable, the obtained translation from PCF to the untyped lambda calculus is executable in Coq. For instance, we can translate the PCF term negating boolean terms as follows:

9.11 Code:

Eval `compute in`

```
(PCF_ULC_c ((fun t => False)) tt (ctype _
  (Lam (condB ' @@ x_bool @@ fff ' @@ ttt '))))).
= Abs (Abs (Abs (Abs (3 @ 2 @ 1))) @ 1 @ Abs (Abs 1) @ Abs (Abs 2))
```

Here we use infix “@@” to denote application of PCF, and `x_bool` is simply a notation for a de Bruijn variable of type `Bool` of the lowest level, i.e. a variable that is bound by the `Lam` binder of PCF in above term.

10. CONCLUSIONS AND FURTHER WORK

We summarize the contributions of this thesis and discuss further work.

10.1. Contributions

We have proved an initiality result for *simply-typed syntax* equipped with *reduction rules*. The category-theoretic iteration principle obtained through the universal property of initiality is sufficiently general to allow for the specification of translations from the term representation to languages typed over *different* sets of sorts.

We have characterized binding syntax with a reduction relation — for instance the lambda calculus with beta reduction — as a *relative monad* over the functor Δ (cf. [Ex. 2.85](#)), encoding not only commutativity properties of substitution, but also its *monotonicity* in the first-order argument. By a suitable strengthening of the definition of relative monad in a 2-categorical context, an additional monotonicity property for the higher-order argument of substitution can be assured, cf. [Rem. 2.86](#). We have also carried the definition of *module over a monad* and several constructions of modules over to modules over *relative monads*.

We then have proved several theorems in the proof assistant Coq: firstly, we implemented Zsidó’s initiality theorem [[Zsi10](#), Chap. 6], summed up in this work as a reference in [Sect. 3.2](#). Secondly, we have proved the initiality theorem of [Chapt. 4](#), yielding a tool, which, when fed with a 2-signature (S, A) , provides the syntax associated to S equipped with the reduction relation generated by the inequations of A . Thirdly, we have proved an instance of our main theorem, [Thm. 5.21](#) of [Chapt. 5](#), for the particular 2-signature of the programming language PCF equipped with reduction rules as in [Fig. A.4](#). The representation of the signature of PCF in the monad of the untyped lambda calculus with beta reduction results in an executable translation from PCF to ULC which is certified to be compatible with substitution and reduction in the source and target languages.

10.2. Further Work

In the future, we hope to prove and implement initiality theorems for richer type systems. In particular, *dependent* types and *polymorphism*, two important steps towards certified programs and code reusability, respectively, should be accounted for.

Furthermore, the modelling of semantics should be improved to allow reasoning about important properties such as *termination*.

As mentioned before, the implementation of initiality results in a proof assistant may serve as a framework for research about programming languages and logics. For this reason we envisage the implementation in a proof assistant of [Thm. 5.21](#) in its full generality.

We present these points in detail:

Fine-grained modelling of reduction For a given 2-signature (a signature together with a set of inequations), models of this 2-signature so far were basically functors which associate, to any set “of variables”, a preordered set — intuitively a model of “terms” over the set of variables¹. The preorder \leq on such a model corresponds to the reduction relation on the term model, i.e. the “term” t reduces to t' if and only if $t \leq t'$.

The modelling of reductions via *preorders* may be considered too coarse in several aspects:

- different reductions might lead from one term to another. However, the use of preorders to model reduction does not allow to distinguish two reductions with the same source and target.
- The hard-coded reflexivity rule makes reasoning about *normalization* — in particular *termination* — difficult.

Instead of considering *preordered* sets (indexed by sets of free variables) as models of a 2-signature, it would thus be interesting to consider a structure which allows for more fine-grained treatment of reduction, such as graphs or categories. In other words, we might build models of 2-signatures from relative monads into the category of *graphs* or (small) *categories*. Using this new definition of model, one might then envisage to prove an initiality theorem analogous to the one already proven, and to use the additional structure obtained by switching to graphs or categories to reason about the aforementioned properties.

Inequations, Syntactically Fiore and Hur [\[FH10\]](#) develop a syntactic theory of equations over a higher-order signature, allowing for proofs of soundness and completeness with respect to the models of the signature and the equations. Similar techniques should allow for a syntactic presentation of our *inequations*. Apart from the obvious goal of soundness and completeness, such a syntactic presentation would also facilitate the specification of reductions in the computer implementation in Coq: in particular, it would make it possible to specify reductions without any knowledge about category-theoretic concepts.

¹We ignore the typed case for the moment, which is analogous.

A minimal goal would be to have a data type — dependent on a 1-signature — which allows to specify the usual half-equations, mainly obtained from substitution and from composition of arities, e.g., $\text{app} \circ (\text{abs} \times \text{id})$. To a term of this data type, one could associate a family of morphisms of modules which constitutes the carrier of a half-equation: the algebraic properties (being a morphism of modules, which corresponds to the compatibility of substitution with meta-substitution in [FH10], could be proved once and for all by induction.

More sophisticated type systems New programming languages tend to be equipped with more and more sophisticated type systems: *dependent types* allow to ensure properties of function output and thus secure plugging together of functions. *Polymorphism* allows for the reuse of code in various situations. An algebraic characterization of such sophisticated type systems with variable binding via a universal property is still missing. We hope to extend initiality results to encompass these type systems.

A wider class of arities The present initiality theorems encompass arities, i.e. term constructors, of quite simple nature: the only operations considered are product — for constructors with *multiple* arguments — and context extension, for modelling variable binding.

It would be desirable to consider more general term formers. Hirschowitz and Maggesi [HM12] have introduced a notion of strengthened arity which allows, for instance, to treat a term former of explicit flattening $\mu : T \circ T \rightarrow T$. Ultimately, we hope to find a very general *simple* criterion for arities and signatures for which an initial model can be provided.

A certified research tool The obtained results should — as we have already done for untyped syntax with reductions — be implemented in a theorem prover such as Coq. In this way, an initiality theorem may be used as a practical tool for easily experimenting with different languages. Changing a language would be done by simply changing its specifying signature, whereas all necessary data and properties such as certified substitution and iteration, but also reductions, would be provided by the system. For this computer implementation and suitable reduction rules, it would also be desirable to obtain automatically a reduction *function* r in addition to the reduction relation. This reduction function might be validated against the relation in the sense that one may prove that for any term t , one has $t \leq r(t)$.

A. SYNTAX AND SEMANTICS OF LAMBDA CALCULUS AND PCF

The following section informally introduces the syntax and semantics of PCF and ULC, as it might be introduced in some computer science textbook. Our presentation of the lambda calculus is inspired by Barendregt and Barendsen’s course [BB94], and that of PCF by Hyland and Ong’s paper [HO00].

A.1. Syntax of Lambda Calculus and PCF

Let V be a countably infinite set (of variables). The syntax of ULC is given by

$$\Lambda ::= v \mid \Lambda @ \Lambda \mid \lambda v. \Lambda ,$$

where $v \in V$ varies over variables.

The programming language PCF is a *typed* language, more precisely a *simply-typed* language. It is given by

- a set of *sorts*,
- a set of *terms* and
- a *typing* map associating a sort to any term.

We take the presentation of PCF from Hyland and Ong’s paper on full abstraction [HO00]. The sorts of PCF are constructed from two base sorts and a function type constructor:

$$T_{\text{PCF}} ::= \iota \mid o \mid T_{\text{PCF}} \Rightarrow T_{\text{PCF}} .$$

The *terms* of PCF are defined in two steps: at first, we define a set of *raw terms*, which actually contains more elements than we want. Afterwards, we define a *welltypedness* predicate on those raw terms. The terms of PCF then are the well-typed raw terms. The raw terms of PCF are given by the grammar of Fig. A.1.

Note that we use the same infix notation $_@_$ for application in PCF and ULC. We also write $f(x)$ for $f @ x$ when no confusion can arise. The constants c_A of sort A are the basic constants from logic and arithmetic, i.e. booleans **T** and **F**, natural numbers n , successor and predecessor as well as test for zero, and conditionals. They are listed in Fig. A.2.

| | | | |
|-----|-------|---------------------|----------------------|
| s | $::=$ | \perp_A | undefined |
| | | c_A | constant |
| | | x_A | variable |
| | | $s@s$ | application |
| | | $\lambda x : A.s$ | abstraction |
| | | $\mathbf{Fix}_A(s)$ | fixed point operator |

Figure A.1.: Grammar of PCF

| | | | |
|--------------------------|---|---|------------------------------------|
| n | : | ι | naturals (for $n \in \mathbb{N}$) |
| \mathbf{T}, \mathbf{F} | : | o | boolean constants |
| \mathbf{S} | : | $\iota \Rightarrow \iota$ | successor |
| pred | : | $\iota \Rightarrow \iota$ | predecessor |
| zero? | : | $\iota \Rightarrow o$ | test on zero |
| \mathbf{cond}_ι | : | $o \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota$ | conditional for naturals |
| \mathbf{cond}_o | : | $o \Rightarrow o \Rightarrow o \Rightarrow o$ | conditional for booleans |

Figure A.2.: Constants of PCF

Instead of all raw terms from the definition of Fig. A.1 we only consider *well-typed* terms, that is, those raw terms that are typable according to the typing judgements of Fig. A.3.

| | | |
|---|--|---|
| $c_A : A$ | $\perp_A : A$ | |
| $\frac{M : A \Rightarrow A}{\mathbf{Fix}_A(M) : A}$ | $\frac{M : A_2}{\lambda x : A_1. M : A_1 \Rightarrow A_2}$ | $\frac{M : A_1 \Rightarrow A_2 \quad N : A_1}{M @ N : A_2}$ |

Figure A.3.: Typing rules of PCF

A.2. Semantics of Lambda Calculus and PCF

Functional programming languages such as PCF and ULC allow for *computation* by *reduction*, as explained in Sect. 1.2.6. The prime example of *reduction rule* is the *beta*

rule of ULC,

$$(\lambda x.M)N \rightsquigarrow_{\beta} M[x := N] , \quad (\text{A.2.1})$$

where $M[x := N]$ denotes the term M where free occurrences of the variable x have been replaced by N in a capture-avoiding manner.

The above rule may be considered to “generate” beta reduction in the sense that we also consider

1. reductions in *subterms* such as in $\lambda x.(\lambda y.M)N$ and
2. *chains of reductions*, that is, reductions consisting of multiple steps.

Thus, to be more precise, what is usually called “beta reduction”, is in fact the closure of the relation specified by the rule given in [Disp. \(A.2.1\)](#) under propagation into subterms as well as transitivity and reflexivity, denoted by \rightarrow_{β} in Barendregt and Barendsen’s course [\[BB94\]](#). In general we associate three different relations to any set of reduction rules, see [Sect. 1.2.6](#).

Reduction in PCF is given by a beta rule similar to [Disp. \(A.2.1\)](#) and several additional reduction rules concerning the fixed point operator and the logical and arithmetic constants. We list them using a small-step semantics as given in [\[HO00\]](#) or in Pitts’ lecture notes on denotational semantics [\[Pit99\]](#). Analogously to the lambda calculus with beta reduction, we denote by “ \rightarrow_{PCF} ” the reduction relation obtained as closure under propagation into subterms as well as reflexivity and transitivity.

$$\begin{aligned}
&\lambda x : A.M(N) \rightsquigarrow M[x := N] \\
&\mathbf{Fix}(g) \rightsquigarrow g(\mathbf{Fix}(g)) \\
&\mathbf{S}(n) \rightsquigarrow n + 1 \\
&\text{pred}(0) \rightsquigarrow 0 \\
&\text{pred}(\mathbf{S}(n)) \rightsquigarrow n \\
&\text{zero?}(0) \rightsquigarrow \mathbf{T} \\
&\text{zero?}(\mathbf{S}(n)) \rightsquigarrow \mathbf{F} \\
&\mathbf{cond}_{\sigma}(\mathbf{T})(M)(N) \rightsquigarrow M \quad (\sigma \in \{o, \iota\}) \\
&\mathbf{cond}_{\sigma}(\mathbf{F})(M)(N) \rightsquigarrow N \quad (\sigma \in \{o, \iota\})
\end{aligned}$$

Figure A.4.: Reduction rules of PCF

BIBLIOGRAPHY

- [ABF⁺05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2005.
- [ACU10] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads Need Not Be Endofunctors. In C.-H. Luke Ong, editor, *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2010.
- [Acz93] Peter Aczel. Galois: A Theory Development Project. Technical Report for the 1993 Turin meeting on the Representation of Mathematics in Logical Frameworks., 1993.
- [Ahr11] Benedikt Ahrens. Modules over relative monads for syntax and semantics. 2011. To be published in Math. Struct. in Comp. Science, [arXiv:1107.5252](https://arxiv.org/abs/1107.5252).
- [Ahr12] Benedikt Ahrens. Extended Initiality for Typed Abstract Syntax. *Logical Methods in Computer Science*, 8(2):1 – 35, 2012.
- [AR99] Thorsten Altenkirch and Bernhard Reus. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999.
- [AZ11] Benedikt Ahrens and Julianna Zsidó. Initial Semantics for higher-order typed syntax in Coq. *Journal of Formalized Reasoning*, 4(1):25–69, September 2011.
- [BB94] Henk Barendregt and Erik Barendsen. Introduction to Lambda Calculus. [ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf](http://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf), 1994. revised 2000.
- [BHKM11] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning*, pages 1–19, 2011. 10.1007/s10817-011-9219-0.
- [Bir35] Garrett Birkhoff. On the Structure of Abstract Algebras. In *Proc. Cambridge Phil. Soc.*, volume 31, pages 433–454, 1935.

- [BM98] Richard S. Bird and Lambert Meertens. Nested Datatypes. In Johan Jeuring, editor, *LNCS 1422: Proceedings of Mathematics of Program Construction*, pages 52–67, Marstrand, Sweden, June 1998. Springer-Verlag.
- [BP99] Richard S. Bird and Ross Paterson. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.*, 9(1):77–91, 1999.
- [CAA⁺86] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
- [CF09] Venanzio Capretta and Amy Felty. Higher-order abstract syntax in type theory. In S. Barry Cooper, Herman Geuvers, Anand Pillay, and Jouko Väänänen, editors, *Logic Colloquium 2006*, volume 32 of *Lecture Notes in Logic*, pages 65–90. Cambridge University Press, 2009.
- [Ch1] Adam Chlipala. Certified Programming with Dependent Types. <http://adam.chlipala.net/cpdt/>.
- [Ch10] Adam Chlipala. An Introduction to Programming and Proving with Dependent Types in Coq. *Journal of Formalized Reasoning*, 3(2):1–93, December 2010.
- [Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.
- [Coq10] Coq. The Coq Proof Assistant. <http://coq.inria.fr>, 2010.
- [FH07] Marcelo P. Fiore and Chung-Kil Hur. Equational systems and free constructions (extended abstract). In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 607–618. Springer, 2007.
- [FH10] Marcelo P. Fiore and Chung-Kil Hur. Second-order equational logic (extended abstract). In Anuj Dawar and Helmut Veith, editors, *CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2010.
- [Fio02] Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '02, pages 26–37, New York, NY, USA, 2002. ACM.

-
- [FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, LICS '99*, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society.
- [GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09*, pages 327–342, Berlin, Heidelberg, 2009. Springer-Verlag.
- [GH08] Murdoch J. Gabbay and Martin Hofmann. Nominal renaming sets. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2008.
- [GL03] Neil Ghani and Christoph Lüth. Rewriting via coinserterers. *Nord. J. Comput.*, 10(4):290–312, 2003.
- [GP99] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax Involving Binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, Washington, DC, USA, 1999. IEEE Computer Society Press.
- [GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.
- [GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *J. ACM*, 24:68–95, January 1977.
- [Hir] Tom Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. 19 pages, submitted.
- [HM07a] André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007.
- [HM07b] André Hirschowitz and Marco Maggesi. The algebraicity of the lambda-calculus. *CoRR*, abs/0704.2900, 2007. informal publication; informal publication.
- [HM10a] André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010.
- [HM10b] André Hirschowitz and Marco Maggesi. Nested Abstract Syntax in Coq. *Journal of Automated Reasoning*, pages 1–18, 2010. 10.1007/s10817-010-9207-9.

- [HM12] André Hirschowitz and Marco Maggesi. Initial Semantics for Strengthened Signatures. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, Tallinn, Estonia, 24th March 2012*, volume 77 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–38. Open Publishing Association, 2012.
- [HO00] J. M. E. Hyland and C.-H. Ong. On full abstraction for PCF: I. Models, observables and the full abstraction problem II. Dialogue games and innocent strategies III. A fully abstract and universal game model. *Information and Computation*, 163:285–408, 2000.
- [Hof95] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, Scotland, 1995. <http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-327/>.
- [Hof99] Martin Hofmann. Semantical Analysis of Higher-Order Syntax. In *In 14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.
- [HS98] Gérard Huet and Amokrane Saïbi. Constructive Category Theory. In *In Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. MIT Press, 1998.
- [Hur10] Chung-Kil Hur. *Categorical equational systems: algebraic models and equational reasoning*. PhD thesis, University of Cambridge, UK, 2010.
- [Lei04] Tom Leinster. *Higher Operads, Higher Categories*. London Mathematical Society Lecture Note Series 298. Cambridge University Press, Cambridge, 2004.
- [ML98] Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.
- [MLM92] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic*. Universitext. Springer-Verlag, New York, 1992. A first introduction to topos theory.
- [MS03] Marino Miculan and Ivan Scagnetto. A framework for typed HOAS and semantics. In *PPDP*, pages 184–194. ACM, 2003.
- [O’K04] Greg O’Keefe. Towards a Readable Formalisation of Category Theory. *Electronic Notes in Theoretical Computer Science*, 91:212 – 228, 2004. Proceedings of Computing: The Australasian Theory Symposium (CATS) 2004.

-
- [Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Ewing L. Lusk and Ross A. Overbeek, editors, *CADE*, volume 310 of *Lecture Notes in Computer Science*, pages 772–773. Springer, 1988.
- [Pho93] Wesley Phoa. Adequacy for untyped translations of typed lambda-calculi. In *LICS*, pages 287–295. IEEE Computer Society, 1993.
- [Pit99] Andrew M. Pitts. Lecture Notes on Denotational Semantics. <http://www.cl.cam.ac.uk/teaching/Lectures/dens/>, 1999.
- [Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.
- [Rie93] Jon G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3(4):387–415, 1993.
- [Sim06] Carlos Simpson. Explaining Gabriel-Zisman Localization to the Computer. *J. Autom. Reason.*, 36:259–285, April 2006.
- [SNO⁺10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.
- [SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In César Muñoz Otmane Ait Mohamed and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21th International Conference*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, August 2008.
- [SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.
- [The10] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. <http://coq.inria.fr>.
- [TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: an Introduction*, volume I and II. North-Holland, Amsterdam, 1988.
- [Ven00] Varmo Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, 2000.
- [Zsi10] Julianna Zsidó. *Typed Abstract Syntax*. PhD thesis, University of Nice, France, 2010. <http://tel.archives-ouvertes.fr/tel-00535944/>.